

A Data-Virtualization System for Large Model Visualization

by Christopher Lux

A dissertation submitted in conformity with
the requirements for the degree of Dr. rer. nat.

Bauhaus-Universität Weimar
Fakultät Medien

Abstract

INTERACTIVE SCIENTIFIC VISUALIZATIONS are widely used for the visual exploration and examination of physical data resulting from measurements or simulations. Driven by technical advancements of data acquisition and simulation technologies, especially in the geo-scientific domain, large amounts of highly detailed subsurface data are generated. The oil and gas industry is particularly pushing such developments as hydrocarbon reservoirs are increasingly difficult to discover and exploit. Suitable visualization techniques are vital for the discovery of the reservoirs as well as their development and production. However, the ever-growing scale and complexity of geo-scientific data sets result in an expanding disparity between the size of the data and the capabilities of current computer systems with regard to limited memory and computing resources.

In this thesis we present a unified out-of-core data-virtualization system supporting geo-scientific data sets consisting of multiple large seismic volumes and height-field surfaces, wherein each data set may exceed the size of the graphics memory or possibly even the main memory. Current data sets fall within the range of hundreds of gigabytes up to terabytes in size. Through the mutual utilization of memory and bandwidth resources by multiple data sets, our data-management system is able to share and balance limited system resources among different data sets. We employ multi-resolution methods based on hierarchical octree and quadtree data structures to generate level-of-detail working sets of the data stored in main memory and graphics memory for rendering. The working set generation in our system is based on a common feedback mechanism with inherent support for translucent geometric and volumetric data sets. This feedback mechanism collects information about required levels of detail during the rendering process and is capable of directly resolving data visibility without the application of any costly occlusion culling approaches. A central goal of the proposed out-of-core data management system is an effective virtualization of large data sets. Through an abstraction of the level-of-detail working sets, our system allows developers to work with extremely large

data sets independent of their complex internal data representations and physical memory layouts.

Based on this out-of-core data virtualization infrastructure, we present distinct rendering approaches for specific visualization problems of large geo-scientific data sets. We demonstrate the application of our data virtualization system and show how multi-resolution data can be treated exactly the same way as regular data sets during the rendering process. An efficient volume ray casting system is presented for the rendering of multiple arbitrarily overlapping multi-resolution volume data sets. Binary space-partitioning volume decomposition of the bounding boxes of the cube-shaped volumes is used to identify the overlapping and non-overlapping volume regions in order to optimize the rendering process. We further propose a ray casting-based rendering system for the visualization of geological subsurface models consisting of multiple very detailed height fields. The rendering of an entire stack of height-field surfaces is accomplished in a single rendering pass using a two-level acceleration structure, which combines a minimum-maximum quadtree for empty-space skipping and sorted lists of depth intervals to restrict ray intersection searches to relevant height fields and depth ranges. Ultimately, we present a unified rendering system for the visualization of entire geological models consisting of highly detailed stacked horizon surfaces and massive volume data. We demonstrate a single-pass ray casting approach facilitating correct visual interaction between distinct translucent model components, while increasing the rendering efficiency by reducing processing overhead of potentially invisible parts of the model. The combination of image-order rendering approaches and the level-of-detail feedback mechanism used by our out-of-core data-management system inherently accounts for occlusions of different data types without the application of costly culling techniques.

The unified out-of-core data-management and virtualization infrastructure considerably facilitates the implementation of complex visualization systems. We demonstrate its applicability for the visualization of large geo-scientific data sets using output-sensitive rendering techniques. As a result, the magnitude and multitude of data sets that can be interactively visualized is significantly increased compared to existing approaches.

Kurzfassung

DIE AKTUELLEN ENTWICKLUNGEN im Bereich der Datenerfassungs- und Simulationstechnologien führen zu immer umfangreicheren und komplexeren Datensätzen, die große Anforderungen an moderne Visualisierungssysteme stellen. Dies gilt in besonderem Maße im geowissenschaftlichen Bereich, in dem sehr detaillierte Aufnahmen des Erduntergrunds in immer mehr Gebieten der Erde entstehen. Die Öl- und Gasindustrie ist ein Treiber dieser Entwicklungen, da neue Lagerstätten immer schwerer zu entdecken sind und oft an schwer zugänglichen Orten liegen. Geeignete Visualisierungstechniken sind zum Auffinden der Lagerstätten, deren Erschließung und deren Betrieb unverzichtbar.

Diese Arbeit präsentiert ein vereinheitlichtes Out-of-Core-Daten-Managementsystem sowie spezialisierte Rendering-Verfahren für die interaktive Visualisierung sehr großer geowissenschaftlicher Datensätze aus der Öl- und Gasindustrie. Diese bestehen typischerweise aus einem oder mehreren seismischen Volumen und einer Anzahl zugehöriger Horizonte. Horizonte beschreiben die Grenzflächen zwischen unterschiedlichen Erdschichten und werden üblicherweise als hochaufgelöste Höhenfelder repräsentiert. Dabei kann jede Komponente eines Datensatzes die Kapazität des Graphik- und auch des Hauptspeichers aktueller Computersysteme deutlich überschreiten. Gängige Datensätze liegen heute im Bereich von über 100 Gigabyte und erreichen vereinzelt schon den Terabyte-Bereich.

Die Haupteigenschaft des vorgestellten Daten-Managementsystems ist es, limitierte Systemressourcen für alle Datenkomponenten gemeinsam zu verwalten und bedarfsabhängig zuzuteilen. Das System nutzt hierarchische Quadtree- und Octree-Datenstrukturen für die Level-of-Detail-Präsentation der Datensätze im Graphikspeicher und im Hauptspeicher. Die Auswahl der verwendeten Detailstufen wird durch einen Feedback-Mechanismus gesteuert, welcher Informationen über benötigte Detailgrade direkt während des Rendering-Vorgangs sammelt. Dieser Mechanismus ist dabei in der Lage, ohne die Anwendung kostspieliger Verdeckungsberechnungen, gleichzeitig die Sichtbarkeit von semi-transparenten geometrischen und

volumetrischen Primitiven zu bestimmen. Ein weiteres Ziel des Out-of-Core-Daten-Managementsystems ist eine effektive Virtualisierung von großen Datenmengen. Durch eine Abstraktion der Level-of-Detail-Repräsentationen wird Entwicklern der Umgang mit extrem großen Datensätzen erleichtert, da sie unabhängig von deren komplexen internen Datenrepräsentationen arbeiten können.

Jedes in dieser Arbeit vorgestellte Rendering-Verfahren basiert auf dieser Daten-Virtualisierungsinfrastruktur. Zuerst wird ein effizientes Volumen-Raycasting-System präsentiert, welches eine interaktive Visualisierung mehrerer, sich beliebig überschneidender Volumendatensätze ermöglicht. Basierend auf binärer Raumunterteilung (Binary Space Partitioning) werden die Hüllkörper der einzelnen Volumenprimitive aufgespalten, um Überlappungsbereiche zu identifizieren und mittels angepasster Methoden effizient zu verarbeiten. Für die Visualisierung geologischer Modelle, bestehend aus mehreren, sehr detaillierten Höhenfeldern, wird ein weiteres Raycasting-basiertes Rendering-Verfahren demonstriert. Die Darstellung eines kompletten Stapels von Höhenfeldern geschieht dabei in einem einzigen Rendering-Durchlauf. Eine effiziente Strahlenverfolgung wird durch eine zweistufige Beschleunigungsstruktur realisiert. Diese kombiniert einen Minimum-Maximum-Quadtree für das Überspringen leeren Raumes und sortierte Tiefen-Intervall-Listen in dessen Knoten, um die Schnittpunktsuche auf relevante Höhenfelder und Tiefenbereiche zu beschränken. Abschließend wird ein vereinheitlichtes Rendering-Verfahren für komplette geologische Modelle präsentiert, das die effiziente Darstellung mehrerer, sehr detaillierter Höhenfelder im Kontext sehr großer Volumendaten erlaubt. Die kombinierte Darstellung erfolgt durch ein Raycasting-basiertes Verfahren in einem einzelnen Rendering-Durchlauf, was eine tiefenkorrekte Akkumulation der optischen Beiträge der einzelnen Modellkomponenten ermöglicht. Gleichzeitig erhöht sich die Effizienz der Strahlenverfolgung durch eine starke Reduktion des Verarbeitungsaufwands für potenziell nicht sichtbare Teile des Modells.

Die in dieser Arbeit vorgestellte vereinheitlichte Out-of-Core-Daten-Management- und Virtualisierungsinfrastruktur erleichtert die Implementierung komplexer Visualisierungssysteme deutlich. Dies wird durch die Entwicklung ausgabesensitiver Rendering-Verfahren demonstriert, die erheblich größere und vielfältigere Datensätze auf konventionellen Computersystemen darstellen können als bisherige Ansätze.

Danksagung

S EHR LANGE ZEIT HAT ES GEDAUERT. Viele lange Tage und Nächte sind in die Forschungsarbeit und Fertigstellung dieser Arbeit geflossen. Ohne die moralische, geistige und tatkräftige Unterstützung vieler Personen wäre mir diese Anstrengung nie möglich gewesen. Ich möchte mich bei euch allen aus tiefstem Herzen bedanken!

Besonderer Dank gilt Herrn Prof. Dr. Bernd Fröhlich. Er gab mir die Chance, diese Dissertation in seiner Forschergruppe an der Bauhaus-Universität Weimar zu erarbeiten. Ich fand stets eine offene und vor allem anregende Arbeitsumgebung vor, in welcher immer Raum für meine persönliche Entwicklung und eine freie Ausgestaltung der Arbeit war. Vielen Dank für die vielfältige Unterstützung durch immerwährende Diskussionsbereitschaft, Anregungen und Diplomatie.

Viele meiner Mitarbeiter in der Forschergruppe „Systeme der virtuellen Realität“ sind mir Freunde geworden. Durch ihre unterschiedlichen Charaktere wurde es nie langweilig und meine fachlichen als auch persönlichen Fragen und Probleme fanden immer Gehör. Ich weiß, dass ich oft nicht zu überhören war. Danke an Stephan Beck, Andreas-Christoph Bernstein, Alexander Kulik, André Kunert, Patrick Riehmann und André Schollmeyer für eine sehr angenehme und lockere Arbeitsatmosphäre und dafür, mich auch in mitunter angespannter und gestresster Laune ertragen zu haben.

Ich danke auch den Mitgliedern des Fraunhofer IAIS und des VRGeo-Konsortiums. Sie boten mir eine Anlaufstelle für Fragen bezüglich des mir anfangs fremden Gebiets der Geowissenschaften und unterstützten diese Arbeit durch die Bereitstellung realistischer seismischer Daten. Dabei danke ich insbesondere Manfred Bogen, Thorsten Holtkämper und David D'Angelo für den immer freundlichen und persönlichen Kontakt.

Ein spezieller Dank gilt vor allem meiner Familie. Neben einem stets zugänglichen Rückzugsort bietet ihr mir immer wichtigen Halt. Besonders dankbar bin ich meiner Lebensgefährtin, Angelika Furch, für ihr Verständnis und ihre Liebe gerade in den schwierigen Phasen. Ich freue mich auf das Neue und Spannende, das jetzt kommt.

Data Attributions

THIS WORK WAS SUPPORTED in part by the VRGeo Consortium. In addition to providing helpful insights into geo-scientific workflows, the consortium members kindly allowed us access to real-world seismic data on which we were able to test and demonstrate our research results. We would like to especially highlight Landmark/Halliburton for providing the most interesting and the most challenging seismic data sets, and for allowing us to publish pictures therefrom. In particular, this thesis contains rendered images of data sets from:

- ▶ Norwegian Petroleum Directorate. (www.npd.no/en/)
- ▶ Crown Minerals and the New Zealand Ministry of Economic Development: Taranaki Basin. (www.crownminerals.govt.nz)
- ▶ Wytch Farm oil field by courtesy of British Petroleum, Premier Oil, Kerr-McGee, ONEPM, and Talisman.
- ▶ Gullfaks oil field by courtesy of BHP, Statoil, and ExxonMobil.

Contents

1	Introduction	1
1.1	Geo-scientific Data	2
1.2	Motivation	4
1.2.1	Visualizing Massive Seismic Data	5
1.2.2	Multi-Volume Data	6
1.2.3	Stacked Horizon Height Fields	6
1.2.4	Research Challenges	8
1.3	Contributions	9
1.4	Organization	11
2	Fundamentals and Related Work	13
2.1	Volume Visualization	13
2.1.1	Volume Data	14
2.1.2	Direct Volume Rendering	17
2.1.3	Real-Time Volume Rendering	20
2.2	Height-Field Visualization	23
2.2.1	Height-Field Data	23
2.2.2	Height-Field Rendering	25
2.3	Visualizing Large Data	27
2.3.1	Computer Architecture Considerations	27
2.3.2	Data Set Reduction Strategies	30
2.3.3	Multi-Resolution Rendering	34
2.3.4	Multi-Volume Rendering	39
2.3.5	Out-of-Core Data Management	40
2.3.6	Texture Virtualization	42
2.4	Summary	44
3	Data Virtualization	47
3.1	Problem Setting	47
3.1.1	Problem Analysis	49
3.1.2	System Requirements	51

3.2	System Overview	52
3.3	Multi-Resolution Data	56
3.3.1	Data Representation	56
3.3.2	Data Preparation	59
3.3.3	Data Indexing and Storage	62
3.4	Texture Virtualization	65
3.4.1	The Virtual Texture System	66
3.4.2	Texture Abstraction	70
3.5	Working Set Generation	79
3.5.1	Unified Selection Algorithm	81
3.5.2	Implicit Page Pre-Fetching	83
3.5.3	Page-Priority Generation Approaches	84
3.6	Level-of-Detail Feedback	85
3.6.1	Process Overview	88
3.6.2	Feedback Data	89
3.6.3	Concurrent Generation of Linked Feedback Lists	91
3.6.4	Feedback Evaluation	96
3.7	Out-of-Core Data Management	103
3.7.1	Parallel System Design	104
3.7.2	Out-of-Core Data Fetching and Caching	105
3.7.3	Asynchronous Data Updates and Rendering	107
3.8	Practical Results	109
3.8.1	Data-Page Size Ramifications	109
3.8.2	Per-Pixel Feedback Lists	117
3.8.3	Bindless Textures	121
3.9	Summary	122
4	Rendering of Virtualized Seismic Data Sets	125
4.1	Multi-Volume Ray Casting	125
4.1.1	Problem Setting	126
4.1.2	Ray Casting Virtualized Volumes	128
4.1.3	Multi-Volume Decomposition and Rendering	129
4.1.4	Results	133
4.1.5	Conclusions	136
4.2	Ray Casting Stacked Horizons	138
4.2.1	Problem Setting	138
4.2.2	Acceleration Structure Considerations	141
4.2.3	Ray Casting of Stacked Virtualized Height Fields	144

4.2.4	Results	149
4.2.5	Conclusions	153
4.3	Combined Rendering of Seismic Models	153
4.3.1	Problem Setting	154
4.3.2	Ray Casting Virtualized Volumes	157
4.3.3	Combined Ray Casting of Seismic Models	161
4.3.4	Results	164
4.3.5	Conclusions	168
4.4	Summary	169
5	Conclusions	171
5.1	Summary	171
5.2	Future Work	175
5.3	Outlook	179
	Bibliography	181
	Curriculum Vitae	195
	Ehrenwörtliche Erklärung	197

Chapter 1

Introduction

VISUALIZATION is a fundamental step in the process of transforming data into knowledge. Scientific visualizations are primarily focused on the visual display of spatial data originating from measuring or simulating physical objects or processes. The fundamental purpose of such visualizations is to enable scientists and engineers the ability to gain insight and understanding from the geometry and topology of their data sets [Gal95]. Interactive three-dimensional visualizations are created through real-time computer graphics, allowing users to investigate complex data in ways not possible in pure numerical form.

Current scientific visualization systems are struggling with an ever growing problem: *data explosion*. This term describes the phenomenon in which data sets are becoming too large in order to be handled conveniently. Through technical advancements in data acquisition and simulation technologies, more data is recorded in much higher resolution and quality. The data sizes are still continuing to outpace even the exponential development of computing and visualization hardware as predicted by Moore's law. To overcome the limitations of current hardware systems, so-called out-of-core rendering algorithms in combination with level-of-detail techniques are required. Such approaches are specially designed to effectively reduce the amount of data loaded and processed without compromising the final rendering result.

Throughout this thesis we particularly focus on the development of real-time out-of-core rendering methods for the visualization of extremely large geological and seismic data sets on conventional computer systems. These data sets are composed of different data primitives such as large volumetric data sets and highly-detailed surface models describing geologic properties and structures in the earth's subsurface. The ever growing scale of the data sets, ranging from hundreds of gigabytes up to multiple terabytes in

size, in combination with growing demands for visual quality, presents very challenging research problems for interactive visualization systems.

In the following sections we first provide an overview of the properties and characteristics of geoscientific data sets (Section 1.1), followed by a description of the particular challenges and motivations related to the visualization of large geological models (Section 1.2) and end by describing the research contributions of this thesis (Section 1.3).

1.1 Geo-scientific Data

Seismic surveys are generated using the principles of reflection seismology [SG95]. Which represents the key technology when exploring large subsurface regions to identify hydrocarbon reservoirs and assess reservoir potential and compartmentalization without even breaking ground. During this process seismic data is acquired by emitting sound waves into the ground and recording the amplitude and travel times of the reflected sound, a procedure very similar to medical ultrasound diagnostics. The recorded data is then processed using seismic migration processes in order to create an accurate seismic survey depicting the magnitude of seismic wave-reflections in a block of the earth's subsurface.

While two-dimensional seismic surveys are still commonly used for on-shore oil fields, three-dimensional reflection seismology almost entirely replaced two-dimensional seismic imaging methods for off-shore oil fields. Two-dimensional surveys are sets of 2D images representing seismic data recorded along linear paths extruded into the ground. These vertical slices of the subsurface are called *seismic lines* and are generally recorded in a coarse grid pattern to examine a larger region. On the other hand, three-dimensional surveys are a much more powerful tool for subsurface exploration as they represent a very fine-grained and complete 3D reflection volume of the earth's subsurface; essentially, they constitute a large stack of vertically aligned seismic lines.

In Figure 1.1 a small section of a volumetric seismic survey is shown illustrating the raw reflection amplitude values as well as a basic direct volume rendering (DVR) visualization of the data set. The shown volume rendering visualization uses a common color and opacity mapping, assigning blue and red values to extreme negative and positive amplitude values, while smoothly transitioning over white in between. Similarly, high opacity

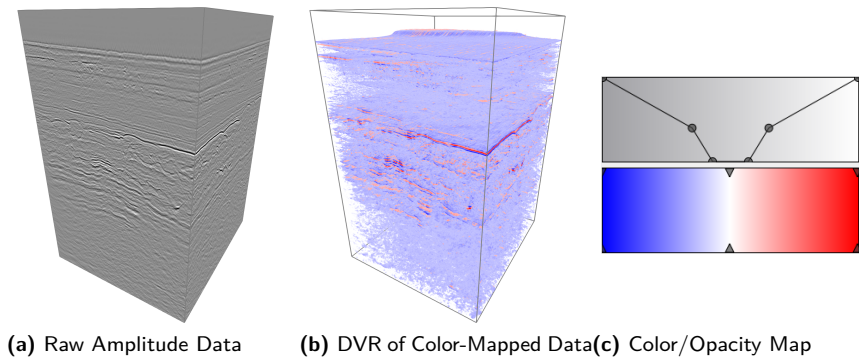


Figure 1.1: A section from a three-dimensional seismic survey. (a) raw recorded reflection amplitude values. (b) direct volume rendering (DVR) of surveyed section using a typical color and opacity map shown in (c).

values are assigned to the extreme amplitude values, while transitioning over complete transparent values. This color and opacity mapping creates a three-dimensional visualization depicting the strongest seismic reflection events in the volumetric data set as pseudo-surfaces.

Geologists and geophysicists utilize interactive visualizations of the seismic volumes during the seismic interpretation process in order to gain insight into stratigraphic properties of the surveyed region. Based on the recorded data, they try to create a geological model of the most important subsurface structures. In the oil and gas industry, hydrocarbon reservoir simulations require realistic geological models as input in order to predict the behavior of the rocks under various recovery scenarios. Considering that an actual reservoir can only be developed once, errors in the interpretation stage have the potential to result in very expensive yet wasted bore-holes.

In order to create realistic geological models, the geologists and geophysicists initially use specialized simulation and seismic processing methods to generate additional volumetric data attributes describing various data properties such as reflection frequency, coherence, dip and potentially many others [RS05]. Using this additional data in their visualizations and simulations they are able to identify particular geologic structures that potentially could trap oil and gas deposits. Through this process the geological model

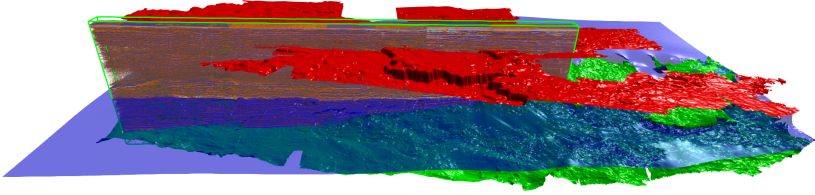


Figure 1.2: Seismic data set containing three different horizons extracted from a common volumetric survey. While the upper most horizon (red) is spatially isolated, the lower horizons (green and blue) are partially overlapping.

is formed as the sum of all the generated and extracted data, providing a description of geometrical and physical properties of the subsurface region of interest.

One fundamental part of geological models are the so called *horizons*. They represent the interface between layers of different materials in the ground. They are represented as surface geometries inside the originating seismic volumes, which act as a frame of reference for the complete model. Figure 1.2 shows a data set containing three horizons and a subsection of their generating seismic volume.

Successful exploration of an oil field may also depend on different data primitives supplemental to the derived and interpreted data primitives from the initial survey. Such supplemental data generally introduces contextual information into the models and contributes to a better understanding of large-scale data sets. Typical additional data primitives include 2D seismics from early exploration surveys, well data, production data, potential field data, image data from satellites or maps and many others. The combination of the multiple cumulated volumetric and geometric geologic data with the many disparate data types constitutes very complex data assets employed in the oil and gas industry.

1.2 Motivation

The interactive visualization of large seismic surveys and geological models has already been firmly established for data analysis and interpretation processes in the oil and gas industry. It enables geo-scientists to gain insight into and explore their data in ways that were not possible compared to

traditional methods, in which only a single seismic line at a time could be handled [LB98, SMG03]. Interactive volume visualization techniques enhance visual pattern recognition, thereby enabling a much more intuitive and direct identification of geologic structures and potential hydrocarbon reservoirs. Beyond the use for analysis processes, three-dimensional interactive visualizations are also heavily utilized for communicating or discussing interpretation results during planning and decision-making processes.

1.2.1 Visualizing Massive Seismic Data

The requirements for seismic visualization systems in the oil and gas industry are rapidly outgrowing the capabilities of the current state of the technology. Improvements in seismic acquisition technologies are delivering more data with much higher resolution and quality. This technological progress leads to data set sizes growing much faster than the development of underlying computing and visualization hardware [Owe07]. Especially the local texture memories of today's graphics processing units (GPU) used for storing and accessing the seismic data during the rendering process are several orders of magnitude smaller than recent data sets. In order to deal with this inundation of data, simplification and out-of-core rendering approaches need to be applied, thus reducing the amount of data that is actually loaded, processed and visualized.

On the other hand, especially within the oil and gas domain, large display and immersive projection-based systems are employed for collaborative purposes and most importantly for the exploration of large geologic structures in the full context of all surrounding data in large-scale surveys. Furthermore, the recent availability of very high-resolution display and projection systems allows users to very accurately preserve and visualize subtle features of the high-resolution source data [CN10]. It has been shown that displays reaching or even surpassing the visual acuity of the human perceptual system can significantly enhance effectiveness of the visualizations because of the fact that additional or more accurate data can be displayed [YHN07]. This dramatic increase of usable pixel density adds to the demands on the underlying visualization systems for very efficient and especially scalable rendering techniques.

Using data simplification methods to deal with large-scale data sets which in turn need to be displayed on expanding display systems with increasing pixel densities in order to present more detailed views onto the data may

seem contradictory. However, when comparing the data set sizes to the actually presentable information, the data sets remain multiple orders of magnitude larger than what recent high-resolution displays can effectively convey. For instance, an 8k display as used in [CN10] offers an absolute resolution of 33.5 megapixels, while a medium sized volumetric seismic survey consists of multiple hundreds of billion samples by today's standards. Even individual horizon surfaces extracted from such a survey are comprised of up to hundreds of millions of samples. Although volume data is mostly displayed translucently, meaning that multiple data samples are contributing to a single pixel, the presentable amount of data still vastly exceeds the capability of such displays. Consequently, data simplification and out-of-core rendering methods are strong requirements for future visualization systems as the data set sizes are still continuing to outgrow even the explosive development of display, computing and especially local memory systems.

1.2.2 Multi-Volume Data

The oil and gas industry is continuously improving the seismic coverage of subsurface regions in existing and newly developed oil fields. Individual seismic surveys are large volumetric data sets, which have precise coordinates in the Universal Transverse Mercator (UTM) coordinate system. It is common practice that the many seismic surveys acquired in larger areas are re-sampled and merged into a single large data set. Figure 1.3 shows a data set created from 33 individual surveys. The various sub-surveys have different resolutions, different orientations and are partially or even fully overlapping due to reacquisition during oil production. The harsh, visible edges in the shown data set stem from potentially differing recording methods and accuracies.

Re-sampling and merging of individual, only partial spatially overlapping, surveys into a single volume grid is very undesirable, yet is the most common solution currently utilized. Reasons include the extended pre-processing times, concerns regarding numerical inaccuracies and data bloat resulting from up-sampling operations on data with differing original resolutions.

1.2.3 Stacked Horizon Height Fields

During the interpretation process, horizon surfaces are extracted from the seismic volume by expert-guided tracking mechanisms. These surfaces gen-

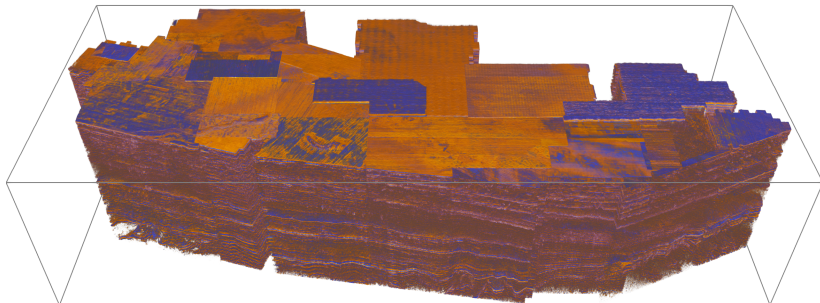


Figure 1.3: Large volumetric seismic data set produced by re-sampling and merging 33 smaller surveys.

erally exhibit a height-field like topology. The horizon surfaces are typically visualized by triangulating the complete height fields and rendering the resulting mesh using traditional rasterization methods. The ever increasing size of seismic surveys generates extremely large horizon geometries composed of hundreds of millions of points. Figure 1.2 shows a typical geological model consisting of three very large stacked horizons. A single horizon in the shown model is composed of 120 million samples, resulting in approximately 240 million triangles and many gigabytes of raw geometry data. Traditional rendering techniques are not able to display these meshes interactively because the geometry data exceeds the available memory as well as the geometry throughput of current graphics hardware.

Rendering approaches designed for traditional terrain visualization purposes can be adapted to seismic data sets containing multiple horizon height fields, such as continuous level-of-detail triangulation techniques [DWS⁺97, LKR⁺96, LP01] and geometry compression and out-of-core streaming methods [LP02, DSW09]. However, horizon height-field data differs in important aspects from elevation data used in traditional terrain rendering. Horizon models often do not represent a closed surface. For example, they exhibit holes in fault regions or areas of missing data. Furthermore, geological models typically contain multiple horizon surfaces stacked on top of each other. Hence, additional problems arise from the depth complexity introduced by occlusions and intersections of individual horizons in the data sets. Occluded parts of the scene could be omitted from the rendering process; on the one hand reducing the amount of data to be processed and

on the other hand potentially lowering the rendering workload to improve rendering times.

1.2.4 Research Challenges

In the oil and gas domain, seismic data sets represent collections of a number of different data primitives. In this work we concentrate on the development of rendering methods for their two most important components: the volumetric seismic surveys and the horizon height fields. The presented efforts aim at widely available computer platforms as dedicated high-performance visualization hardware gradually vanished with the success of commodity graphics processing units (GPUs).

The fundamental problem in dealing with large data primitives is foremost their sheer size. Every individual part of a seismic model may exceed the bandwidth and size limits of CPU-bound and GPU-bound memory, necessitating advanced out-of-core visualization algorithms. When dealing with such large models, achieving real-time rendering performance requires methods for carefully managing bandwidth requirements, controlling working set sizes and ensuring coherent data access patterns. While existing rendering algorithms are potentially able to visualize individual components of a seismic model, the efficient simultaneous visualization of a multitude of model components remains a research challenge.

The handling of multiple arbitrarily overlapping volumes while avoiding costly resampling processes can vastly improve the flexibility and efficiency of the visualization of multiple seismic surveys. While existing multi-resolution volume rendering methods are able to handle single large volumes [LHJ99, BNS01, PTCF02], this problem, however, has not yet been fully addressed. Effective identification of overlapping volume regions in such models is essential for efficient rendering strategies to avoid costly oversampling of single-volume regions and to enable correct intermixing of multi-volume segments.

Likewise, in order to efficiently visualize complex geological models containing stacked horizon surfaces, special rendering methods are required. These methods need to consider data occlusion as well as the special height-field characteristics of the horizon data. Traditional rasterization methods cannot render such data sets in real-time and general terrain rendering approaches [PG07] have not been adapted to deal with the specific structure of multiple horizon layers. Furthermore, with increasing screen resolutions

and screen-space errors below one pixel, the geometry throughput of current GPUs has become a major performance bottleneck. Dealing with complex stacked geologic models requires costly occlusion culling methods as proposed in [PGSF04]. A central aspect of our work is the exploration of output-sensitive rendering strategies, such as ray casting, for these visualization problems. Output-sensitive rendering approaches can provide much improved rendering performance while inherently dealing with data visibility [DSW09, DKW09]. However, none of the existing ray casting-based rendering systems are capable of efficiently visualizing complete stacks of highly detailed, mutually occluding and overlapping horizons contained in large geological models.

The ultimate goal of this work is the creation of a visualization system for subsurface data capable of interactively visualizing entire geological models. The essential feature of such a system is the combined rendering of highly detailed stacked horizon surface geometries and massive volume data. Thereby it is paramount to account for data occlusions not only of a single primitive type, but globally over volume as well as height-field geometries. There are multiple reasons for this, but the most important one is to effectively steer the data-selection for the out-of-core data management of the different underlying data primitives. If, for example, large parts of horizons are covered by an opaque volume, the occluded parts of the data set can be omitted. Thus, precious memory resources can be saved and potentially committed to other parts of the model. Equally important is the development of a combined rendering method that treats such models unified in order to only access actually visible data. Currently no infrastructure exists for the efficient out-of-core management and rendering of geometry and volume data.

1.3 Contributions

The overarching goal of this thesis is to provide an advanced rendering system capable of efficiently visualizing large geological models composed of large volumetric data primitives and height field-like surface geometries on commodity hardware. In this context we restrict ourselves to data primitives defined on regular or Cartesian grids, the most common representations for three-dimensional volume data as well as the two-dimensional representation of height-field surfaces. These representations also fit the regular structure

of texture maps, a major data resource available for GPU-based rendering techniques.

Data Virtualization In this thesis we present a general out-of-core data management system supporting geological models consisting of multiple large volume and height-field data sets, where each data set may exceed the size of the graphics memory or even the main memory. Based on multi-resolution representations of the source data sets, this system efficiently shares available system memory resources among all currently loaded primitives. We employ multi-resolution methods based on hierarchical octree and quadtree data representations, allowing us to adaptively represent the data at different local resolutions. The actual selection of the appropriate resolution for the individual parts of the data sets can thereby depend on a number of factors including the existing memory limits, the visibility of the data and the effectively required resolution. The central goal of this out-of-core data management system is the virtualization of large data sets, facilitating easy access to the data without special knowledge of the internal data layouts. This system represents the base infrastructure for the individual rendering systems presented in this work.

Multi-Volume Rendering We propose an efficient GPU-based volume ray casting system for the rendering of multiple arbitrarily overlapping multi-resolution volume data sets. Through the use of shared data resources in system and graphics memory we are able to support a virtually unlimited number of simultaneously visualized volumetric data sets. We demonstrate how efficient volume virtualization allows for multi-resolution volumes to be treated exactly the same way as regular volumes. Binary space partitioning (BSP) volume decomposition of the bounding boxes of the cube-shaped volumes is used to identify the overlapping and non-overlapping volume regions. The resulting volume fragments are extracted from the BSP tree in front-to-back order for rendering. This approach requires recomputations of the BSP tree only if the spatial relationship of the volumes changes.

Rendering of Layered Height Fields We developed a ray casting-based rendering system for the visualization of geological subsurface models consisting of multiple highly detailed height fields. The visualization of an entire stack of height-field surfaces is accomplished in a single rendering pass using a two-level acceleration structure for efficient ray intersection computations. This structure combines a minimum-maximum quadtree for empty-space skipping and sorted lists of depth intervals to restrict ray

intersection searches to relevant height fields and depth ranges. Our shared out-of-core data management system virtualizes the access to the individual height fields, allowing us to treat the individual surfaces at different local levels of detail (e.g. occluded horizon parts are represented at a much lower resolution). A feedback mechanism is employed during rendering which directly generates level-of-detail information for updating the cut from the multi-resolution hierarchies on a frame-to-frame basis.

Combined Visualization of Volume and Layered Height Field Data We ultimately present a unified rendering system for the visualization of entire geological models consisting of highly detailed stacked horizon surface geometries and massive volume data. The combined visualization of seismic volumes and interpreted stacked horizon data is accomplished by using a ray casting-based rendering technique. This approach allows to inherently deal with occlusions between different data types in a combined visualization of volume, polygonal and image-based primitives such as textured horizon height-fields. This rendering system is based on the unified out-of-core data-virtualization system. The virtualization abstracts complex multi-resolution data layouts from the logical data representations and thus allows access to the data without regard for its physical storage. Expanding upon the feedback mechanism used for the preceding horizon stack rendering technology, we generate per-pixel feedback lists during the actual rendering process. These lists contain level-of-detail feedback information for horizon as well as volume data sets. This allows us to inherently deal with occlusions between different data types in a combined visualization of volume and image-based primitives, such as horizon height-fields.

The combination of level-of-detail feedback and output-sensitive rendering techniques enables the realization of rendering systems employing truly demand driven out-of-core data management and level-of-detail data selection, without the application of potentially costly occlusion culling techniques.

1.4 Organization

This thesis is organized into three parts. The first part (Chapter 2) provides an overview of fundamental concepts and techniques and reviews and discusses related and considered work with respect to this thesis.

The second part (Chapter 3) focuses on the design and realization of an out-of-core data-virtualization system. We start by outlining the specific

problem setting of scientific visualizations in the geo-scientific domain. After providing an overview of our conceptual system design, we detail its most important components, including the employed multi-resolution data representations, the data virtualization approach and the feedback mechanism gathering data-usage information during a rendering process. The chapter is concluded by the presentation of practical results of a prototypical system implementation.

The third part (Chapter 4) then presents rendering approaches for particular visualization problems encountered in the context of large geo-scientific data sets. The proposed rendering approaches include the following: a multi-volume ray casting-based rendering approach for the efficient visualization of multiple arbitrarily overlapping volumes; a ray casting-based rendering method for the direct visualization of multiple layered height-field surfaces; and ultimately a combined rendering approach for the visualization of entire geological models consisting of highly detailed layered height-field surfaces and massive volume data.

This thesis is concluded in Chapter 5 with a summary of the presented contributions and an overview of possible future work in relation to the application of the proposed techniques as well as directions and inspiration for future research.

Chapter 2

Fundamentals and Related Work

THE goal of this thesis is to provide advanced rendering methods capable of efficiently visualizing today's and most importantly tomorrow's large-scale geological models on commodity computer systems. The presented contributions build upon related work from different domains spanning scientific visualization and real-time rendering methods. This chapter provides an overview of fundamental concepts and rendering techniques related to volume rendering (Section 2.1) and to the visualization of height-field surface geometries (Section 2.2). Section 2.3 reviews related and considered work for advanced rendering techniques attempting to adapt to available system resources using multi-resolution and out-of-core techniques. The concluding Section 2.4 discusses the relationships between the prior discussed related work and the techniques contributed by this work.

2.1 Volume Visualization

In the broad field of scientific visualization, one very essential discipline is the rendering of volumetric data. The concept of volume rendering essentially describes the generation of images from explicitly three-dimensional, hence, volumetric data. The interactive visualization of large volumetric data sets has many diverse application areas including medicine, aeronautics and geosciences. Medical diagnostics and surgery planning make use of visualizations of highly detailed volume data acquired by Computed Tomography (CT) and Magnetic Resonance Imaging (MRI) scanners. In aeronautics numerical simulations, such as Finite Element Methods (FEM) or Computational Fluid Dynamics (CFD), generate vast amounts of volume data depicting properties of complex dynamic systems. For the purpose of identifying hydrocarbon reservoirs, the oil and gas industry is utilizing

seismic imaging as well as simulation methods to generate volume data sets depicting the structure of the earth’s subsurface as outlined in Section 1.1.

2.1.1 Volume Data

In the domain of scientific visualizations, volumetric data sets are primarily defined as sampled representations of continuous three-dimensional scalar functions:

$$f(\mathbf{x}) \in \mathbb{R} \quad \text{with} \quad \mathbf{x} \in \mathbb{R}^3 \quad (2.1)$$

These functions may represent data based on measurements or simulated data. There exist different representations for these data sets, such as unstructured, irregular point sets. However, uniform rectilinear grids are typically used to store the sampled function since they expose a regular data topology and geometry. The regular volume data representation inherently defines the sample locations through fixed grid-origin and grid-spacing parameters [Kau94]. In contrast to unstructured data representations, data sample lookups can be accomplished in constant time making this type of volume representation the most commonly used. The uniform grid representation is the type of volume representation on which the methods described in thesis are based.

Volume data acquired by scanning physical objects or by simulating complex systems using numerical techniques typically results in three-dimensional arrays of values. Analogous to the term pixel for picture element, the individual elements of the discretized volume data sets are called *voxels*, short for volume elements. They can be envisioned as small cubes constituting the complete data set, as illustrated in Figure 2.1a. However, in practice the data samples are obtained at infinitesimally small points located at the vertices of a uniform grid centered at the voxel cubes [HKERS02] (cf. Figure 2.1b). In real-time applications, the reconstruction of a continuous scalar field or function (Equation 2.1) from the discrete sample locations is typically performed using box or linear interpolation filters. The box filter only calculates the nearest-neighbor interpolation, resulting in sharp discontinuities between neighboring voxels. The trilinear interpolation, however, represents a good trade-off between computational cost and smoothness of the reconstructed data. Today, fast trilinear interpolations of volume data are typically performed directly during rendering by the intrinsic texture-filtering functionality of current GPUs.

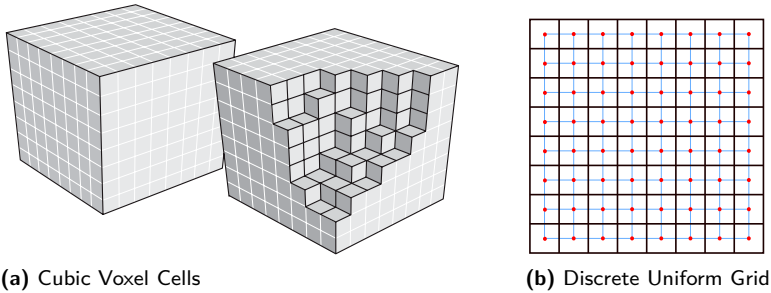


Figure 2.1: Different conceivable perceptions of discretized volume data defined by a uniform grid. (a) shows volume data illustrated as cubic cells (image from [EHK⁺06]). (b) shows volume samples (voxels) as infinitesimally small points (red) at the vertices of the uniform grid (blue) at the center of cubic cells (black).

Classification

An essential step in the process of visualizing a volumetric data set containing an abstract spatially varying physical attribute, is the assignment of optical properties like color and opacity to the voxel data. Through this *classification* process, particular objects or features of interest contained in the data are visually separated from surrounding and non-relevant data. Classification in the wider sense tries to extract objects from the data as sets of associated voxels. However, in volume visualization, classification refers to the process of finding an appropriate mapping of voxel values to optical properties [LCN98].

In real-time volume visualization, *transfer functions* are the fundamental tool of transforming the abstract data values to displayable properties. The most common form of transfer functions are direct one-dimensional mappings from the domain of the input voxel values into RGBA tuples for color and opacity. They are typically implemented as simple lookup-tables stored in 1D-textures on the GPU. During the rendering process they are then directly evaluated upon sampling the raw volume data from the actual data set. Figure 2.2 shows the application of different simple transfer functions to a CT data set of a human torso, illustrating the use of the opacity mapping by revealing different levels of insight into internal

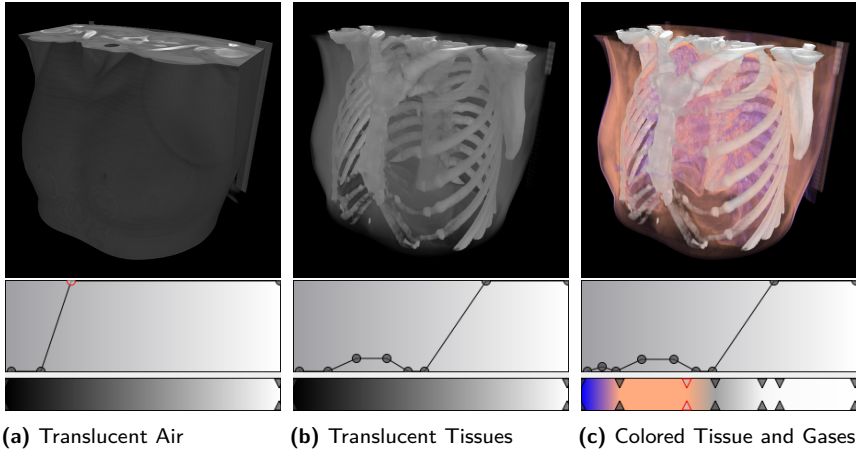


Figure 2.2: Different transfer functions applied to a CT data set of human torso. (a) uses grayscale color mapping and assigns translucent values only to low densities (gases). (b) reveals bone structure by assigning translucent values to mid densities (tissue). (c) shows skin in orange and soft tissue in blue, revealing the lungs.

organs. Extending upon the use of basic color and opacity mappings, multi-dimensional transfer functions take local derivatives of sample values into account, allowing to more effectively distinguish material boundaries from homogeneous volume regions [KD98, KKH02].

Multi-Attribute and Multi-Volume Data

Part of this work is concerned with the rendering of *multi-volume* data sets (cf. Section 1.2.2). The fundamental characteristic of such data sets is that they are composed of multiple volume grids of different resolutions and different spatial orientations, which are partially or even fully overlapping. In the literature, the differentiation of *multi-volume* and *multi-attribute* data sets can be quite unclear. Multi-attribute data sets contain multiple completely spatially aligned volume grids representing different physical properties for each sampling position. In some cases, spatially aligned volumes portraying different grid resolutions are already considered multi-

volume data. Combining medical scans from CT with MRI machines is one such example of this [RTF⁺06]. Through the continuous reconstruction of the source signals, it can be argued whether these data sets are indeed just multi-attribute data sets.

2.1.2 Direct Volume Rendering

Volume rendering methods describe the process of transforming volumetric data sets into two-dimensional images [LCN98]. All volume rendering approaches can be classified into two categories: indirect and direct volume rendering. Indirect approaches visualize the contents of volume data sets by explicitly extracting and rendering surface geometries corresponding to certain properties of objects potentially contained in the data set. This is generally accomplished by employing contouring algorithms to generate surfaces corresponding to a specific iso-value [LC87, KBSS01]. In contrast, direct volume rendering (DVR) methods visualize the contents of volumetric data sets directly without intermediate surface geometries corresponding to particular objects or features of interest [DCH88, HHS93, Max95].

To generate an image from the optical properties assigned to the abstract volumetric data sets, direct volume rendering requires a model of how light interacts with light. Physical phenomena such as light emission, absorption, scattering or occlusion are described in optical models. They describe the actual transfer of light through the volume data, which acts as a participating medium. Particular optical models with varying levels of physical realism exist to illustrate different features of the actual data [HHS93, Max95]. The most commonly used model in real-time volume rendering is the *emission-absorption* model, which considers the volume data as physical particles that only emit or absorb light. The scattering or refraction of light is not considered in this model, so light always moves in straight lines through the volumes.

Volume Rendering Integral

The light transfer through a volume data set generating a displayable intensity according to the emission-absorption optical model is described

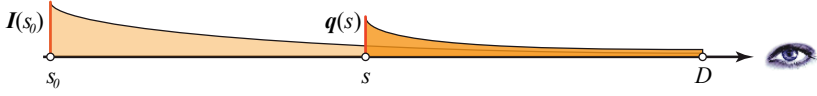


Figure 2.3: This image illustrates the integration of light along viewing rays through a volume between the points s_0 and D according to the volume rendering integral shown in Equation 2.2. (similar to image from [HKERS02])

by the following integral, generally called the *volume rendering integral*:

$$I(D) = I(s_0)e^{-\tau(s_0,D)} + \int_{s_0}^D q(s)e^{-\tau(s,D)} ds \quad (2.2)$$

$$\text{with} \quad \tau(s_1, s_2) = \int_{s_1}^{s_2} \kappa(s) ds$$

It basically describes the integration of light between the points s_0 and D along viewing rays through the volume. Whereby s_0 represents the point where the ray enters the volume at the back and D represents the exit point at the front towards the viewer (cf. Figure 2.3). The first term of the equation describes the light entering the volume data set $I(s_0)$ attenuated by the volume. The integration of the light that is emitted ($q(s)$) and absorbed inside the volume is described by the second term. The term $\tau(s_1, s_2)$ refers to the *optical depth*, which is a measure of how quickly light gets absorbed inside the translucent volume. It is based on the optical properties assigned to the volume samples, described by the absorption coefficient $\kappa(s)$. The transparency of the material between two points along the viewing rays is therefore defined by the following term:

$$T(s_1, s_2) = e^{-\tau(s_1, s_2)} = e^{-\int_{s_1}^{s_2} \kappa(s) ds} \quad (2.3)$$

Discrete Evaluation

The volume rendering integral can generally not be solved analytically. It is typically approximated by subdividing the integration domain into n discrete intervals. The integral can therefore be represented by a Riemann sum [EHK⁺06]:

$$I(D) = \sum_{i=0}^n C_i \alpha_i \prod_{j=1}^{i-1} (1 - \alpha_j) \quad (2.4)$$

In this equation the product $C_i\alpha_i$ represents the intensity of the i -th sample $I(s_i)$ along the viewing ray based on the optical properties of color (C_i) and opacity (α_i) assigned by the transfer function. These values are assumed to be constant over the discretized intervals. To avoid problems due to the interpolation of opacity-weighted colors [WMG98], the colors assigned by the transfer functions are explicitly not pre-multiplied by their associated opacities.

The iterative evaluation of the volume rendering integral during the volume rendering process is typically called *compositing*. There are basically two ways to evaluate the integral: *front-to-back* and *back-to-front* compositing. As suggested by their names, the difference between these two methods is the order in which samples along the viewing rays are integrated. Under the assumption of constant distances between the discrete sample locations, the following incremental calculation schemes ensue. Front-to-back compositing:

$$\begin{aligned} C_{dst} &= C_{dst} + (1 - \alpha_{dst})C_i\alpha_i \\ \alpha_{dst} &= \alpha_{dst} + (1 - \alpha_{dst})\alpha_i \end{aligned} \tag{2.5}$$

Back-to-front compositing:

$$C_{dst} = (1 - \alpha_i)C_{dst} + C_i\alpha_i \tag{2.6}$$

The variables C_{dst} and α_{dst} hold the intermediate results during the calculation, resulting in the final intensity that is eventually displayed for that particular viewing ray. The back-to-front compositing scheme does not require the storage and update of an opacity variable because the contribution to the resulting output intensity is only dependent on the current sample color and opacity. While this helps to save a certain amount of memory during the computation process, front-to-back compositing allows for optimizations that highly outweigh these savings. Most importantly, the ability to stop the iterative process when the accumulated opacity along particular viewing rays reaches levels that do not allow any more intensity to be accumulated. This optimization is referred to as *early-ray-termination* [Lev90].

In certain cases it might be necessary to change the distances between the discrete sampling locations (e. g. when using adaptive sampling to accelerate the volume rendering process). The discrete approximation in Equation 2.4

assumes constant sample distances. Changing the sampling distances would result in an incorrect visualization of the volume. However, it is possible to account for varying sampling interval lengths by correcting the opacities according to the ratio of the new ($\Delta\tilde{s}$) to the initial interval (Δs) lengths [LCN98]:

$$\tilde{\alpha} = 1 - (1 - \alpha)^{\frac{\Delta\tilde{s}}{\Delta s}} \quad (2.7)$$

2.1.3 Real-Time Volume Rendering

Volume rendering is a very active field of research with much work focused on real-time volume rendering techniques. Engel et al. [EHK⁺06] provide a comprehensive overview of volume rendering methods applicable to current graphics hardware. The following sections concentrate on techniques and research most related to the methods presented in this work.

In GPU-based real-time volume rendering, generally two basic approaches exist: texture slicing and ray casting. Both fundamentally implement a numerical evaluation of the volume rendering integral (Equation 2.2) by making use of the capabilities of the graphics hardware to store the uniform volume grid in 3D textures. This allows for re-sampling the volume grid using fast trilinear filtering inherent to such texturing hardware. The difference lies in how the actual sample positions are generated.

Volume Texture Slicing

Texture slicing-based direct volume rendering methods implemented on graphics hardware, first introduced by Cullip and Neumann [CN93] and improved upon by Cabral et al. [CCF94], use a stack of typically viewport-aligned planar surfaces as proxy geometry in order to re-sample the volume data. After the rasterization of these planes, the generated volume samples are converted to color and opacity values. Depending on the technological state of the available hardware, the application of the current transfer function is achieved by using either a dependent texture lookup into a 1D texture or by using color index modes on less powerful GPUs. Finally, the resulting values are blended into the frame buffer in front-to-back or back-to-front order. Further work presents improvements to the visual quality and efficiency of the basic texture slicing approach [WE98, RSEB⁺00, EKE01]. This method represents the most widely used rendering approach for the

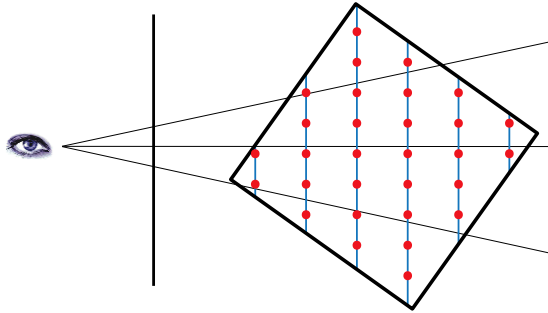


Figure 2.4: This image depicts how slice-based volume rendering samples the volume data using viewport-aligned polygonal planes. This approach generates inconsistent sampling distances, illustrated by three viewing rays.

visualization of small-sized volumetric data sets before GPUs allowed to more freely sample 3D textures without relying on complex proxy geometries.

As an object-order method, the density and therefore the amount of slice planes that must be generated and rendered depends directly on the data complexity as well as the desired output quality. These texture-slicing approaches basically evaluate the volume rendering integral for a set of viewing rays in a parallel lock-step fashion. Figure 2.4 illustrates the texture-slicing approach. It also conveys a major drawback of this approach under perspective projections. The sampling distances are not constant over the generated image. For viewing rays farther from the center of projection the sampling distance increases, thereby creating potentially incorrect intensities for these image regions.

Volume Ray Casting

Ray casting methods are a very direct implementation of a discrete evaluation of the volume rendering integral. First introduced by Levoy [Lev88, Lev90], ray casting works by sampling the volume data at discrete locations along rays originating at the viewer position. Figure 2.5 illustrates how for each pixel, a ray is generated and traversed through the volume. Note that when using this approach, the sampling distances are identical for each ray passing through the volume. Typically, ray casting is performed in front-to-back order enabling the easy extension of the basic algorithm

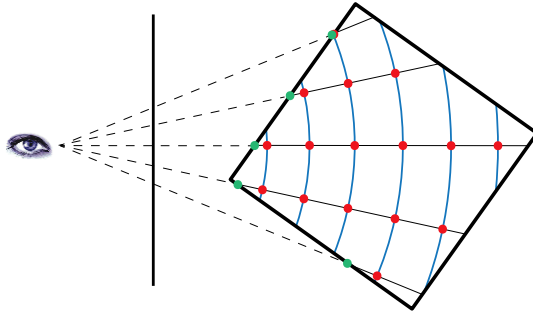


Figure 2.5: This figure illustrates volume ray casting. Individual viewing rays are traversed and sampled at a constant sampling distance.

with advanced acceleration techniques such as adaptive sampling, early ray termination or empty-space skipping [DH92, YS93].

With the evolution of the fixed-function texture mapping functionality in early graphics hardware generations to much more freely programmable shading functions in recent GPUs, the direct implementation of the ray casting algorithm became feasible. Early GPU-based volume ray casting approaches [RGW⁺03, KW03, WKME03] are dependent on multiple render passes and temporary image buffers for storing intermediate results. Due to limitations on the execution of conditional branches and limited loop counters of the available GPU generations at that time, they had to rely on multi-pass algorithms to perform the ray traversal. Despite these restrictions, Krüger et al. [KW03] demonstrated how to leverage the early-z optimizations inherent to the GPUs in order to implement early ray termination to optimize for better memory bandwidth usage and save computation on invisible parts of the volume. With the evolution of GPUs, the implementation of the complete ray casting algorithm in a single rendering pass is presented including very easy early ray termination [Sch05, SSKE05]. The algorithm is therefore implemented in a single shading program, expressing the volume traversal of a single viewing ray. The execution of this program for each pixel covered by the volume is triggered by the rasterization of the bounding box geometry of the data set. Scharsach [Sch05] demonstrated the implementation of an empty-space skipping scheme by refining the coarse proxy geometry to more closely fit the non-transparent volume data and

thereby generating the volume entry position much closer to the actually visible data. Klein et al. [KSSE05] proposed a different approach to empty-space skipping by exploiting frame-to-frame coherence. They reprojected the depth buffer from the previous frame to generate the entry points for the volume ray casting.

On current generations of GPUs, volume ray casting is now the best performing volume rendering technique. The flexibility of the independent traversal of individual rays through volume data sets makes it superior to slice-based rendering approaches. Furthermore, GPU-based volume ray casting has virtually no proxy geometry processing overhead, whereas the texture slicing techniques need to generate and rasterize vast amounts of polygonal proxy geometries. The volume rendering methods presented in this work therefore purely rely on ray casting approaches.

2.2 Height-Field Visualization

The visualization of surface geometries described by height-fields has many important applications, such as terrain visualization for flight simulations, military battlefield visualization and geographic information systems (GIS). Figure 2.6 illustrates the use of a height field representing elevation data for terrain visualization. Beyond this most prominent application, height-fields are also widely used in the movie and recently the game industries as displacement maps to add detail to coarser polygonal geometries [Coo84]. Depending on the field of application, height fields are referred to with different names: *elevation maps* in terrain visualization and displacement maps when describing surface details. In the scope of this work, height-field like geometries are employed to represent highly detailed horizon surfaces in larger composite geological models (cf. Section 1.2.3).

2.2.1 Height-Field Data

A height field is defined as a two-dimensional scalar field. The contained scalar values are interpreted as a displacement orthogonal to the defining base plane. They therefore are defined as sampled representations of continuous two-dimensional scalar functions embedded in three-dimensional space:

$$f(\mathbf{x}) \in \mathbb{R} \quad \text{with} \quad \mathbf{x} \in \mathbb{R}^2 \quad (2.8)$$

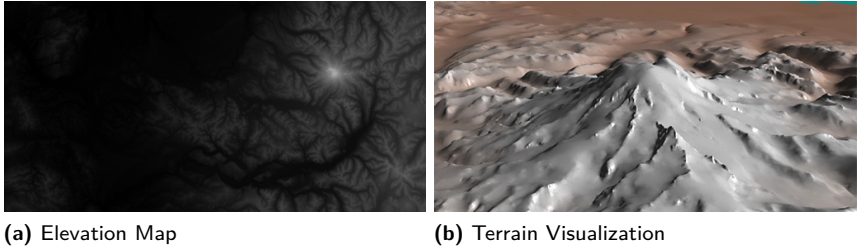


Figure 2.6: Illustration of a three-dimensional terrain surface described by a two-dimensional height field. (images from [LP01])

The continuous function is in practice discretized into a finite amount of data samples. Analog to volume data described in Section 2.1.1, the most commonly used representation of the discrete data set is a uniform rectangular grid with a defined regular topology and geometry (cf. Figure 2.6a). The reconstruction of a continuous scalar field is typically performed by bilinear interpolation filters. In real-time rendering, techniques implemented on graphics hardware perform the reconstruction employing the available fast texture filtering units.

Because the displacement information typically is the sole information stored in height-field data sets, dependent information needs to be derived directly from them. For instance, the calculation of local illumination effects of the represented surface require normal vectors $\hat{\mathbf{n}}$, which can be calculated using the partial derivatives of Equation 2.8 as follows:

$$\mathbf{n} = \begin{pmatrix} -\frac{\partial f}{\partial x} \\ -\frac{\partial f}{\partial y} \\ 1 \end{pmatrix} \quad \hat{\mathbf{n}} = \frac{\mathbf{n}}{\|\mathbf{n}\|} \quad \text{with} \quad \mathbf{n}, \hat{\mathbf{n}} \in \mathbb{R}^3 \quad (2.9)$$

Layered Height-Field Data

As height fields describe orthogonally displaced surfaces, they cannot represent closed geometries. However, by using multiple layered height fields, closed objects can be described in multiple depth layers [PO06]. This means that, for example, closed three-dimensional objects can be described by two spatially aligned height fields: one representing the front and another repre-

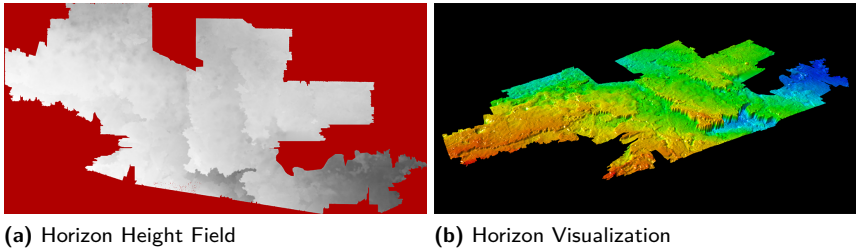


Figure 2.7: Seismic horizon represented by a partial height field data set. (a) shows the height field encoded surface data with red color indicating missing data. (b) shows a rendering of the horizon height field with pseudo colors mapped to the surface.

senting the back surface. The purpose of such approaches is to represent objects with highly-detailed surfaces, which would require extremely fine tessellated polygonal meshes to reproduce the detail otherwise.

The structure of multiple spatially aligned height fields also applies to the stacked horizon height fields contained in the geological models, which are a primary focus of this work. The fundamental difference is that the stacked horizon models do not represent closed objects. They represent multiple partial surfaces stacked on top of one another (cf. Section 1.2.3). Furthermore, the individual horizon layers do not represent complete height field surfaces, as data can be missing due to insufficient seismic coverage or faults in the seismic volume. Figure 2.7 shows a single horizon height field with the missing data highlighted in red color. During the rendering of such partial height fields, it is important to properly discard the missing areas in the data.

2.2.2 Height-Field Rendering

Approaches for the rendering of height-field data sets basically fall into two categories: triangulation approaches and direct ray casting of the height map. The triangulation methods generate a polygonal mesh which is rendered using traditional rasterization. A height map with the grid resolution N generates an amount of triangles in the order of N^2 . This quickly results in meshes too large to be handled without the application of

level-of-detail techniques. Many real-time rendering approaches designed for traditional terrain visualization employing continuous level-of-detail triangulation techniques were proposed to reduce the amount of displayed triangles [DWS⁺97, LKR⁺96, LP01, LP02]. However, such object order approaches exhibit large computational overhead when creating view-dependent triangulations. With increasing screen resolutions and screen-space errors below one pixel, extremely small polygons are generated, which GPUs are not able to efficiently rasterize. Consequently, the geometry throughput of current GPUs is becoming a major performance bottleneck and output-sensitive rendering strategies such as ray casting provide much improved rendering performance for the visualization of height-field surfaces [DSW09, DKW09].

The direct visualization of height-field surfaces using ray casting-based methods is a very active and well explored field of computer graphics research. Early CPU-based methods, primarily targeted at terrain rendering applications, were based on a 2D-line rasterization of the projection of the ray into the two-dimensional height-field domain to determine the cells relevant for the actual intersection tests [Mus88, CS93, CORLS96, QQZ⁺03]. A hierarchical ray casting approach based on a pyramidal data structure was proposed by Cohen and Shaked [CS93]. They accelerated the ray traversal by employing a maximum-quadtrees, storing the maximum height-displacement value of areas covered by its nodes in order to identify larger portions of the height field not intersected by the ray.

Enabled by the rapid evolution of powerful programmable graphics hardware, texture-based ray casting methods implemented directly on the GPU were introduced. The first published approach by Qu et al. [QQZ⁺03] still uses a line rasterization approach similar to the traditional CPU-based approaches. The relief-mapping [OBM00] methods pioneered by Policarpo et al. [POC05, PO06] employ a parametric ray description combined with an initial uniform linear search which is refined by an eventual binary search restricted to the found intersection interval. Considering that the initial uniform stepping along the ray may miss high-frequency details in the height field, these methods are considered approximate. While these initially published GPU-based ray casting algorithms are not utilizing any kind of acceleration structures, later publications proposed different approaches. Donnelly [Don05] described the use of distance functions encoded in 3D-textures for empty-space skipping. The drastically increased texture memory requirements make this technique infeasible for the visualization of large height fields. Later methods proposed by Dummer [Dum06] and

Policarpo et al. [PO06] exhibit drastically reduced memory requirements. They calculate cone ratios for each cell to describe empty space regions above the height field allowing for fast search convergence during the ray traversal. The very high pre-computation times of these techniques only allow for the handling of quite small height-field data sets. This problem was addressed in the subsequent publications by Oh et al. [OKL06] and Tevs et al. [TIS08]. They built upon the traversal of a maximum-quadtrees data structure on the GPU akin to the algorithm presented by Cohen and Shaked [CS93]. The idea of encoding the quadtree in the mipmap hierarchy of the height field results in very moderate memory requirements.

Policarpo et al. [PO06] introduced a method for handling a fixed number of layered height fields without any acceleration structures in a single rendering pass. They simply move along the ray using a fixed sampling step and intersect all the height fields at once using vector operations on the GPU, which works efficiently for at most four height fields encoded in a single texture resource.

The methods proposed in this thesis rely on output sensitive height-field ray casting techniques. Their close relationship to volume ray casting methods facilitate the efficient combined visualization of horizon height fields embedded in a seismic volume.

2.3 Visualizing Large Data

Ever increasing data set sizes pose a major problem for scientific visualizations. The data sets exceed the size of fast local memory available to the CPU and the GPU, the two major processing units in today's computer architectures. Naïve or brute force rendering algorithms are understandably not able to interactively visualize large data sets because the complete data sets can only be stored on slower external storage. Methods ensuring coherent data access while managing working set sizes and bandwidth requirements are key for achieving real-time rendering of large data sets [GKY08].

2.3.1 Computer Architecture Considerations

The design of current computer architectures forms a memory hierarchy with two principle yet contradicting characteristics: memory size opposed

to memory access latency and bandwidth. A fundamental observation is that memory closer to the actual processing units is fast but small, and memory farther away in the hierarchy becomes bigger but only allows slow access. Figure 2.8 illustrates the architecture and memory hierarchy in current computer systems. The graphics subsystem is located on a separate device, which is connected to the host by the system bus. External storage is massive in size, but only allows for slow transfers to the host with very high data access latencies (i.e. the time delay between a data read request and its availability). Data loaded into the host's main memory can be accessed much faster, but the size of this memory is currently limited to tens of gigabytes. For the actual rendering, the data is required to be resident in the memory directly connected to the GPU (VRAM). This memory is usually one order of magnitude smaller than the main memory, but at the the same time represents the fastest memory in the system.

A typical data flow in a large model rendering system involves all stages of the memory hierarchy. The data is initially loaded from external storage. It is then potentially preprocessed in main memory to fit certain requirements of the graphics device. It then is uploaded to the device through the system bus and is eventually processed by the GPU to generate the final rendered image.

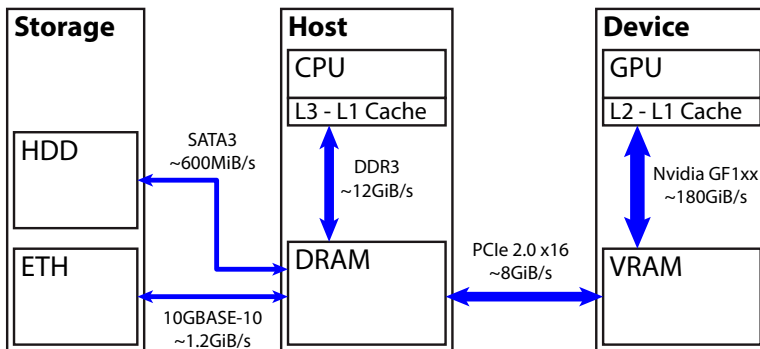


Figure 2.8: The basic architecture and memory hierarchy of a current computer system. Fast access to data during rendering is only possible using device embedded VRAM. Data is loaded to the graphics device through the narrow PCIe bus system.

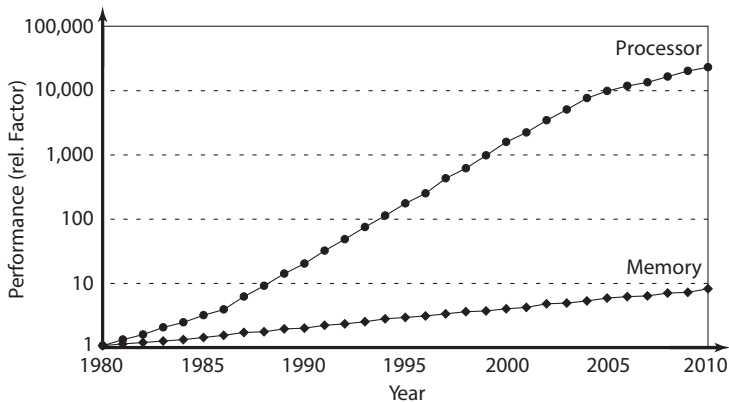


Figure 2.9: The growing disparity between processor performance and memory access speed by example of CPU and DRAM performance over the last three decades. (similar to image from [HP06])

Another problem of current computer hardware architectures is that the performance of the processing units grew and still grows much faster than the speed of the memory. This disparity between processor performance and off-chip memory access times is typically called the *memory wall* [WM95]. In Figure 2.9 this performance gap is illustrated over the last three decades using the example of CPU and DRAM performance. The same issue is present between the GPU and its connected VRAM. As a result, large data computations and especially visualizations are inherently limited by memory bandwidth rather than by processing. To absorb much of this performance disparity, current processing units feature extremely fast but ultimately very small on-chip caches, thus extending the memory hierarchy by at least another stage.

Modern GPUs allow two views of memory with different access characteristics. Memory can be addressed and accessed in a common linear fashion employing a two-level cache for general purpose computations. However, memory can also be accessed as texture memory. Textures represent structured, two- or three-dimensional, spatially addressed arrays in memory. These arrays are typically not stored in linear fashion. They are stored using particular spatial organizations to ensure efficient access to neighboring texture samples. The efficiency of repeated access to a certain local

region around a texture sample, as with the calculation of local gradients, is improved by special texture caches. Very importantly, textures are solely accessed through specific *texture filtering units*, dedicated to perform fast sample interpolations. The volume and height-field rendering methods presented in this work make use of these special features by storing uniform grids of the source data as textures in GPU memory.

2.3.2 Data Set Reduction Strategies

Fast and efficient access to the data can only be part of the solution for the visualization of massive data sets, as the local memory available to the GPU can only store a fraction of the whole data model, and loading and rendering the data little-by-little will never generate real-time results. The memory external to the GPU is still orders of magnitudes too slow to deliver the data quickly enough. Modern GPUs generate the best rendering performance when processing mostly static data. This means that data uploaded into graphics memory is used for the rendering of multiple frames and should not be changed on a frame-by-frame basis. Data upload through the narrow system bus to update the currently rendered data would stall the rendering pipeline and hence degrade the visualization performance. Therefore, a subset of the original data needs to be defined that, for given display parameters (e.g. viewing position or transfer function), can visually represent the entire data model as closely as possible. This *working set* is updated if it no longer represents an adequate approximation. Determining the working sets is realized based on a combination of several different techniques and data structures for visibility and detail culling.

Visibility Culling

A fundamental class of techniques in computer graphics is concerned with *visibility culling*. By determining which parts of a scene are actually visible, invisible parts can be discarded from the rendering process. More importantly, it allows for the elimination of such parts from the working set and hence from graphics memory. Visibility culling methods are generally classified as from-point and from-region algorithms [COCSD03].

From-region approaches determine what parts of the scene or data set are visible for a given region of space, commonly referred to as a potentially visible set (PVS). The generation of these sets is very computationally

expensive and is typically performed in a pre-process. During rendering, visibility sets might be used to predict what portions of the scene will be possibly visible in the near future depending on the movement of the viewer between adjacent regions of space. This predictive information can be exploited to load data ahead of time into graphics memory. However, regarding the scope of this work, such from-region algorithms are not applicable to volume data sets. The visibility of volume sub-regions heavily depends on the user selected opacity transfer function. Changes to this function immediately invalidate any preprocessed PVS data and require costly recomputations.

From-point visibility culling techniques are applied directly during runtime and determine visibility information depending on the current viewer position. These techniques can further be categorized into object-space and image-space approaches. While object-space approaches determine visibility directly from the geometrical representations of scene objects, image-space approaches rely on the discretization of objects in screen space. A classic example of object-space techniques is *view-frustum culling*. Through view-frustum culling, portions of the data sets that fall outside of the field of view can be quickly discarded. The application of hierarchical data structures further increases the efficiency of this culling process [Cla76, AM00].

Much more complex than these quite basic tests is the determination of portions of the scene that are occluded by others. The visualization of models of high depth complexity especially benefits from occlusion culling techniques. Image-space from-point techniques are widely used in real-time rendering exploiting special GPU features that allow to query the fragment count generated by the rasterization of individual objects. They are mostly applied in hierarchical approaches to amortize the query costs for larger groups of objects [BWPP04, HB04, BHP07]. In the context of volume rendering, various occlusion culling approaches exist. Guthe et al. [GS04] employed a software ray casting algorithm to determine visibility information at a lower resolution than used in the connected GPU-based volume renderer. Gobbetti et al. [GM05] used hardware occlusion queries to cull invisible parts in a hybrid surface and volume rendering framework for massive 3D models. They more recently proposed a pure, direct volume rendering technique based on a screen-space partitioning scheme to interleave the queries with the ray casting-based volume rendering [GMG08].

While hardware occlusion queries are a powerful tool for visibility determination, they can create significant overhead when applied to a large

number of objects. They require data readbacks from the GPU containing the individual query results, which if not carefully used will result in expensive pipeline stalls of the GPU. Furthermore, occlusion queries are bound to the actual viewport resolution of the renderer, prohibiting a subsampling for more aggressive visibility estimations. Crassin et al. [CNLE09] proposed a mechanism to track sub-volume visibility directly during rendering without application of occlusion queries. They use auxiliary render targets to store spatially and temporally subsampled visibility information.

Simplification - Detail Culling

For the visualization of large data sets, especially geologic models, it is not sufficient to solely rely on visibility culling techniques to construct suitable working sets. Large seismic models contain very high-resolution data depicting extremely fine details. However, much of this detail is potentially invisible, mainly caused by data occlusions and the discrete nature of display devices. Through the perspective projection of the data sets into the viewport, larger voxel or height-field blocks might be represented by only a small number of output pixels.

The size of the working set used to represent the complete data set can therefore be further reduced by employing *level-of-detail* techniques. Lower detail representations of large scene portions are employed in their place when they only represent small, distant or otherwise unimportant parts of the data set. Therefore, the memory size as well as the rendering cost for these parts are drastically reduced without significant loss in the quality of the visualization. Mesh simplification algorithms are typically applied for polygonal models [Lue01]. In the context of data primitives based large uniform grids, such as the volume and height-field components in geologic models, *multi-resolution rendering* techniques are primarily used to reduce the working set sizes. They make it possible to locally adapt the grid resolution based on different criteria, such as sub-block screen-projection size, viewer distance or data homogeneity. Multi-resolution volume and height-field data representations are the basis for the data reduction and virtualization strategies presented in this work. More details on the basic data structures and level-of-detail selection criteria of the for this thesis considered and related multi-resolution rendering techniques are discussed in Section 2.3.3.

Data Compression

A more general approach to reduce the size of data sets is to use data compression techniques. A smaller memory footprint not only permits larger data working sets in graphics memory, but it also facilitates more efficient data transfers between memory regions. Data is potentially loaded much faster from external storage into the main memory and, of course, subsequently into the graphics memory. Compression techniques are classified into lossy and lossless methods depending on whether they allow the exact reconstruction of the original data from the compressed data.

While excellent lossless compression solutions exist for general use, modern GPUs are restricted to lossy texture compression techniques. Furthermore, the exposed compressed texture formats mostly only apply to 2D textures, and the support for compressed 3D textures is limited to only very basic compression methods. Lossless compression of volume data is predominantly achieved through packing algorithms. They classify blocks in the volume data as empty and non-empty regarding the chosen transfer function. Only the non-empty blocks are then uploaded to the GPU. During rendering, small auxiliary textures are used to map the blocks back to their original domain [Sch05, LH06].

Proposed real-time volume rendering approaches employing lossy compression methods either decompress the data prior to rendering or are decompressing the data on-the-fly directly on the GPU during rendering. Guthe et al. [GS01, GWGS02] presented volume rendering methods based on wavelet representations of the source volume data to encode the differences between different time steps for volume animations as well as the differences between varying levels of detail for large volume data sets. They decompressed the volume data in main memory and uploaded the uncompressed data to the GPU for rendering. Apparently, this approach does not lower the memory footprint of the data in graphics memory. More recent approaches use the computational power of modern GPUs to perform the volume decompression on-the-fly during rendering. Fout et al. [FAM⁺05] showed that texture packing in combination with vector quantization can be implemented directly on the GPU with good performance results. Such approaches, however, forgo the use of the hardware filtering features and require manual interpolation calculations, thereby increasing the computational overhead.

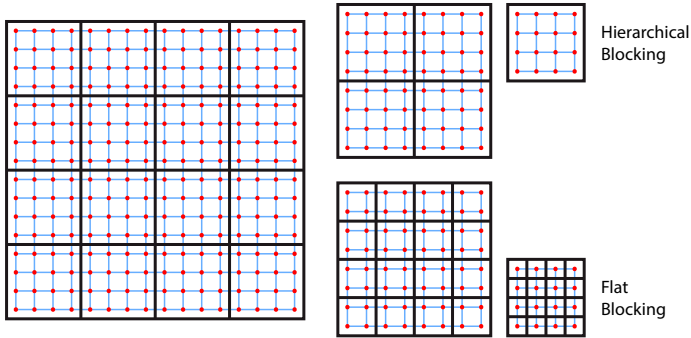


Figure 2.10: Blocking schemes used in multi-resolution volume rendering. Hierarchical approaches adapt the spatial extent of fixed resolution blocks, while flat approaches adapt the resolution of a fixed grid subdivision. (similar to image from [BHMFO8])

In scientific visualization, it is not desirable to introduce errors into the source data through lossy compression methods. This requirement seemingly forbids the use of compression to reduce the graphical working set size. However, when applying level-of-detail approaches to the visualization, which by themselves are just approximations of the source data, compressions could be applied to these coarse data representations.

2.3.3 Multi-Resolution Rendering

Visualizing large seismic data sets requires the use of level-of-detail and multi-resolution techniques to balance between rendering speed and memory requirements. This thesis is concerned with two fundamental components of geological models: the seismic volume and the interpreted horizon surfaces. Both are based on and represented by rectangular uniform grids. Therefore, very similar multi-resolution representations can be applied to the underlying data arrays.

Multi-Resolution Blocking Schemes

Multi-resolution data representations are traditionally applied to the visualization of large volumetric data sets. Such approaches are categorized

as hierarchical or flat blocking schemes. Both schemes utilize the same basic premise of breaking down the original data set into smaller fixed-size blocks, typically called *bricks* in volume rendering. Hierarchical blocking schemes then use these blocks as leaf nodes to create an octree hierarchy [LHJ99, WWH⁺00, BNS01, PTCF02, GWGS02]. Coarser resolutions are represented through inner nodes, which are generated bottom-up by down-sampling eight neighboring nodes from the next finer level. The root node eventually represents the entire data set at its coarsest resolution. Inner nodes have the same size as their child nodes. Consequently, all nodes in the octree are represented by blocks of the same size, as the spatial extent of the blocks is increased. On the other hand, flat blocking schemes keep the spatial extent constant while successively reducing the block resolution [Lju06, BHMFO8].

Figure 2.10 illustrates the difference between the two blocking schemes used in multi-resolution rendering. The flat blocking schemes permit higher empty-block culling rates through their inherent constant classification granularity. However, hierarchical schemes are much more suitable for the visualization of large data sets, because they allow to better adapt the number of volume sub-blocks to the actual resource conditions. When updating the current working set, individual volume blocks can easily be exchanged by others without any memory fragmentation issues. The working set of blocks selected to be stored in graphics memory for rendering

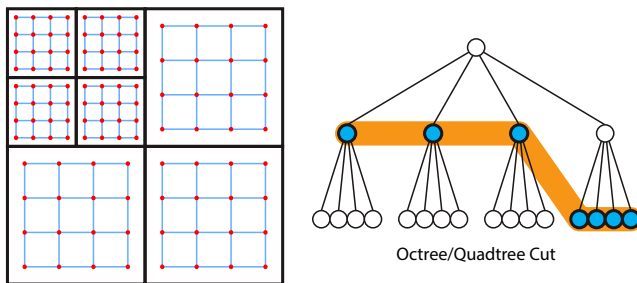


Figure 2.11: The working set of sub-blocks in hierarchical multi-resolutions rendering techniques are represented as a cut from the octree or quadtree hierarchies for three-dimensional volume and two-dimensional image uniform grids, respectively.

is typically represented as a *cut* from the multi-resolution octree hierarchy (cf. Figure 2.11). This cut represents the entire data set at varying local resolutions; this means each part of the data set has an actual representation even though it may not be at the highest available resolution.

Multi-Resolution Volume Rendering

The first hierarchical multi-resolution volume rendering technique presented by LaMar et al. [LHJ99] introduces the use of an octree to represent a large volumetric data set at different local resolutions. They keep the entire hierarchy in main memory and generate the visualized cut based on view-dependent criteria. The selected sub-volume blocks are finally rendered one at a time in back-to-front order using a slice-based volume rendering technique, compositing the individual rendering results in the frame buffer. In contrast, based on a similar octree hierarchy and rendering approach, Boada et al. [BNS01] analyzed the actual volume data to select the used local resolutions based on data homogeneity and user-defined image quality requirements. Weiler et al. [WWH⁺00] improved upon these first approaches by addressing the inter-block interpolation issues present at the boundaries between bricks of different octree levels. Plate et al. [PTCF02] focused on out-of-core resource management in multi-resolution rendering systems. They employed a two-level predictive paging scheme to enable roaming through multi-gigabyte volume data sets by loading only potentially used parts of the volume hierarchy from the hard drive into main memory.

All previous multi-resolution volume rendering approaches share common ground in that they rely on slice-based multi-pass approaches to process and render the volume bricks individually. They compose the rendering results of single volume slices and volume blocks at low frame-buffer precision. The use of high-precision floating point frame-buffer formats can provide a solution, but they drastically increase memory and bandwidth demands. Multi-pass ray casting approaches can compose intermediate results during the ray traversal through individual blocks at high internal precision, but still require high precision formats for the compositing of the entire multi-resolution volume. Despite this composition precision problem, multi-pass approaches have limitations with respect to algorithmic flexibility and rendering quality. For example the implementation of advanced volume ray casting techniques such as early ray termination and empty-space-skipping is cumbersome and inefficient compared to their implementation in a single

pass algorithm. The fundamental problem preventing the implementation of single-pass multi-resolution ray casting techniques is the limitation of modern GPUs to only make it possible to bind a relatively small number of individual texture resources. Therefore, the absolute number of individual volume blocks accessible to a shader program is severely limited.

Consolidating the individual data sub-blocks into a single texture resource allows the implementation of single-pass rendering algorithms. Kraus and Ertl [KE02] described how to use a texture atlas to store the individual volume sub-blocks in a single texture resource. They used an index texture for the translation of the spatial data sampling coordinates to the texture atlas cell containing the corresponding data. Based on this approach, single pass multi-resolution volume ray casting systems were introduced by Gobbetti et al. [GMG08] and Crassin et al. [CNLE09]. Both are based on a classic octree representation of the volume data set and store the octree cut in a 3D-texture atlas. Instead of an index texture to directly address the texture atlas, they use a compact encoding of the octree similar to [LH05]. The leaf nodes of the octree cut hold the index data for accessing the sub-blocks from the texture atlas. Individual rays are traversed through the octree hierarchy using a similar approach to *kd-restart* [HSHH07], which is employed for recursive tree-traversal in real-time ray tracing algorithms on the GPU.

Multi-Resolution Height-Field Rendering

Many real-time rendering approaches designed for traditional terrain visualization employing continuous level-of-detail triangulation techniques were proposed to reduce the amount of displayed triangles [DWS⁺97, LKR⁺96, LP01, LP02]. They create multi-resolution hierarchies based on the triangulated height-field grid. As discussed in Section 2.2.2, with increasing screen resolutions and screen-space errors below one pixel, modern GPUs are limited by their geometry throughput and therefore height-field ray casting techniques can provide much improved rendering performance.

The first multi-resolution-based ray casting approach presented by Dick et al. [DKW09] is able to arbitrarily handle large terrain data sets employing a multi-resolution quadtree representation of the tiled terrain elevation map. They used a screen-space error metric to generate a continuous level-of-detail represented as a cut from the quadtree hierarchy. The height-field tiles corresponding to the selected quadtree cut are then rendered individually in front-to-back order applying a per-tile maximum-quadtrees ray traversal

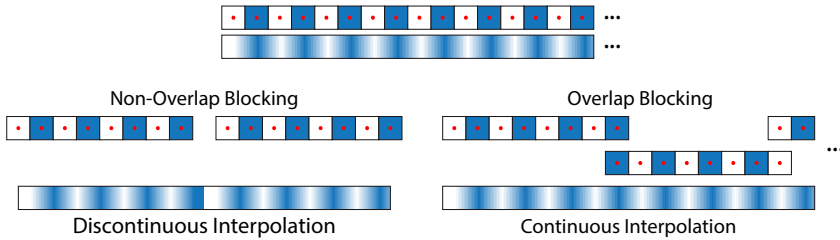


Figure 2.12: Inconsistent interpolation (left) as a result of simple splitting on block boundaries. Through the duplication of data samples at the boundaries (right) correct inner-block interpolations are possible. (similar to image from [EHK⁺06])

based on the works of Oh et al. [OKL06] and Tevs et al. [TIS08]. Further, to prevent fragment overdraw they used an additional rendering pass per tile prior to the actual ray casting to mask out already found height-field intersections, generating additional rendering overhead as it requires multiple render target changes for a single tile.

Multi-Resolution Artifacts

In blocked mixed-resolution rendering approaches, problems arise for the correct interpolation of sample values. With blocks essentially handled as individual uniform sub-grids, the interpolation at the block boundaries is not trivially handled. The left illustration in Figure 2.12 shows the resulting discontinuous interpolation when using a plain blocking scheme. This problem is caused by missing information about neighboring data samples for a consistent interpolation.

LaMar et al. [LHJ99] demonstrated how correct interpolations of blocks of the same level are ensured by explicitly sharing data samples at the boundary between immediately neighboring blocks, as illustrated in Figure 2.12 (right). The size of the shared boundary depends on the application scenario. Plain interpolations require only a one sample wide border, while gradient calculations generally require a two sample wide border between adjacent blocks. This approach allows for easy inner-block linear interpolations at the cost of a small memory overhead to store the boundary samples. Weiler et al. [WWH⁺00] addressed the inter-block interpolation issues present at

the boundaries between bricks of different levels of detail. A smooth and continuous interpolation in such cases is achieved by explicitly copying data samples of adjacent blocks when updating the multi-resolution cut. In addition, they restricted transitions in the hierarchy to differ by at most one level in order to maintain a certain continuity between levels. Ljung et al. [LLY06] and Beyer et al. [BHMF08] conveyed how continuous transitions are achieved for flat multi-resolution blocking schemes without sharing data samples among individual blocks. They generated correct transitions at the boundaries of individual blocks by gathering the required data samples from different blocks and manually computing the interpolated value.

However, the application of these fundamental inter-block interpolation schemes will not conceal all level-of-detail transitions in a multi-resolution rendering approach. The region in which the transition between neighboring blocks occurs is usually very small compared to the actual block size. As a result, the transitions are discernible through quick changes in local data resolution. LaMar et al. [LDHJ00] presented a technique to blend smoothly between adjacent data blocks of varying levels of detail mapped on cutting planes inside a multi-resolution volume. Carmona et al. [CRF09] improved upon this approach by extending its use to full direct volume rendering.

2.3.4 Multi-Volume Rendering

This thesis proposes methods for the visualization of volume data sets consisting of multiple volumes of different resolutions, and different spatial orientations, which are partially or even fully overlapping (cf. Section 1.2.2).

Most research on scenes consisting of multiple volume data sets has been done in the field of medical visualization. Jacq and Roux [JR97] introduced techniques for rendering multiple spatially aligned volume data sets, which can be considered a single multi-attribute volume. Leu and Chen [LC99] made use of a two-level hierarchy for modeling and rendering scenes consisting of multiple non-intersecting volumes. Nadeau [Nad00] supported scenes composed of multiple intersecting volumes. The latter approach, however, requires costly volume re-sampling if the transformation of individual volumes changes. Grimm et al. [GBAG04] presented a CPU-based volume ray casting approach for rendering multiple arbitrarily intersecting volume data sets. They identified multi-volume and single-volume regions by segmenting the view rays at volume boundaries. Plate et al. [PHF07] demonstrated a GPU-based multi-volume rendering system capable of handling multiple

multi-resolution data sets. They identified overlapping volume regions by intersecting the bounding geometries of the individual volumes while also considering the individual sub-blocks of the multi-resolution octree hierarchy. They still relied on a classic slice-based volume rendering method, and thus the geometry processing overhead quickly became the limiting factor when moving either individual volumes or the viewer position.

Roessler et al. [RBE08] demonstrated the use of ray casting for multi-volume visualization based on similar intersection computations. Both approaches rely on costly depth sorting operations of the intersecting volume regions using a GPU-based depth peeling technique. Very recently Lindholm et al. [LLHY09] demonstrated a GPU-based ray casting system for visualizing multiple intersecting volume data sets based on the decomposition of the overlapping volumes using a BSP-tree [FKN80]. This allows for efficient depth sorting of the resulting volume fragments on the CPU. They described a multi-pass approach for rendering the individual volume fragments using two intermediate buffers. While they supported the visualization of multi-resolution volume data sets, their approach is based on the insertion of the volume sub-blocks in the BSP-tree resulting in a very large amount of volume fragments and rendering passes.

2.3.5 Out-of-Core Data Management

Algorithms designed to process data that is too large to fit into the local memory attached to the processing units are generally called *out-of-core* or *external memory* algorithms [Vit01]. Such algorithms specifically manage data transfers between fast internal memory of the CPU and the GPU and slow external storage. They have to consider the different memory bandwidth and latency characteristics of the various layers in the memory hierarchy. Based on the fact that most visualization tasks only require a sub-set of the actual data, most out-of-core rendering algorithms implement some form of application-controlled *demand-paging* [SCESL02]. In multi-resolution rendering approaches, the fixed-size data blocks (e.g. the volume bricks or image tiles) act as *virtual memory pages* - the most basic unit throughout virtual memory management systems.

Virtual memory systems try to make large data sets transparently accessible through multi-level cache hierarchies. Memory pages are moved between the different levels of the memory hierarchy to provide fast access to the currently used parts of the data sets. In real-time out-of-core rendering

approaches, the GPU memory is regarded as the first level cache, containing the current working set of memory pages. The main memory bound to the CPU then acts as a second level cache, storing a super-set of the graphical working set. The external storage finally acts as the *backing store*, holding the entire data set. This *cache hierarchy* allows for the absorption of the high latencies attached to loading parts of the data set from external storage.

During the update of the currently used graphical working set, a *cache miss* is generated when trying to access a memory page not currently present in either cache level. Upon its detection, a request is created to load the particular page from the backing store. In order to prevent stalling of the rendering pipeline while waiting for the requested data pages, the renderer keeps using the coarse versions of the particular portions of the data, which are available through the multi-resolution data representations. The missing data is incorporated into the working set as soon as it is available. Therefore, the rendering system is decoupled from the asynchronously running CPU through a non-blocking demand-driven paging scheme. At this stage, pre-fetching approaches also attempt to predict data pages potentially required in future rendering frames and load them ahead of an actual request from the rendering process.

Typically, working set updates are performed exploiting frame-to-frame coherence. Incremental updates to the multi-resolution hierarchy cuts are possible, because most of the data of a working set does not change between rendering frames. Starting from the currently established cut, greedy-style algorithms perform progressive updates based on certain bandwidth budgets [PTCF02, CF11]. These budgets account for the time it takes to upload data to the GPU memory. This way large changes in the working set are carried out over multiple frames, hence maintaining an interactive visualization by preventing large drops in the frame rate.

Out-of-core rendering approaches based on hierarchical multi-resolution data representations have the advantage of working with fixed memory-page sizes. Each node on every level in the hierarchies portrays the exact same data block size, opposed to flat blocking schemes in which each level contains a different block size. Therefore, memory management is extremely simplified as replacing a page in main memory as well as texture memory is a straight forward operation without any considerations to memory fragmentation whatsoever.

Cache Coherent Data Layouts

With the characteristics of current computer architectures in mind, special data layouts can vastly improve the performance of visualization systems. Data access times dramatically rise if data requested by the rendering method is not resident in the dedicated cache. In order to reduce the amount of such *cache misses* and therefore allow faster data access, *cache coherent* data layouts need to be considered. This basically translates to re-organizing the data in the linear memory with regard to the most frequent data access patterns. Therefore, the probability of a *cache hit* is dramatically increased. The special (and mostly undocumented) memory layouts of textures in GPU memory are a prime example of dedicated data layouts.

However, all other stages in the visualization process of large models also strongly require cache coherent data layouts. For instance, the storage of large data sets on external storage such as large RAID systems exhibits extreme latencies for non-linear access patterns. When trying to read a sub-section of a large volumetric model stored in linear fashion, only extremely small parts of the data can be read sequentially. Reorganizing the data according to a special access pattern and access granularity makes it possible to read sub-sections in sequence and also facilitates pre-fetching of potentially neighboring data blocks.

2.3.6 Texture Virtualization

The work presented in this thesis is focused on large volume and image data sets based on uniform grid representations. For the purpose of rendering, this data is handled as texture resources in the graphics memory of the GPU. The previous sections discussed the use of multi-resolution representations of regular grid data. They mostly rely on rendering strategies that split the geometry of the source data into a large number of small pieces associated with the underlying tiled texture data. These pieces are generally handled one at a time, hence requiring the modification of the rendering algorithms to explicitly attend to the multi-resolution nature of the data sets.

The abstraction of logical texture resources from the underlying data structures is generally referred to as *texture virtualization*. The purpose is to hide the physical data layout and allow access to the data sets without consideration of physical storage. Rendering algorithms therefore can be expressed in such a way that they are mostly unaware of the underlying

multi-resolution representation of the data set. In the context of seismic data sets, this for example makes it possible to map arbitrary multi-resolution volume attributes to horizons, which by themselves are based on multi-resolution height-field images, without additional efforts to possibly maintain several geometry subdivisions for different multi-resolutions assets. This flexibility is highly desired for scientific visualizations in the oil and gas domain.

Virtualization approaches presented for 2D-texture data are based on mipmapping [Wil83] - a technique used to prevent aliasing artifacts by representing texture as a pyramid of images gradually reduced in resolution. During rendering, the optimal mipmap level is calculated per rasterization fragment, which is used to sample the actual texture data. The virtualization approaches for large texture data subdivide the individual mipmap levels of the pyramid into fixed size tiles. The resulting data structure essentially is a multi-resolution quadtree representing the entire texture image. This makes it possible to only partially define particular mipmap levels by removing texture tiles that are currently not required, resulting in the current working set.

The basic problem texture virtualization systems are facing is the estimation of the single texture tiles required to most adequately represent the original texture. Textures are usually mapped to complex non-planar geometries, and therefore can be deformed and stretched with non-uniform geometry parametrization. The actual required parts of the texture pyramid needed for rendering are quite unpredictable and not easily estimated analytically (e. g. through view-dependent criteria).

Tanner et al. [TMJ98] presented the *clipmap*, a unique-texturing approach for the application of high-resolution terrain photographs to digital elevation models, based a subdivided mipmap hierarchy. They relied on a center of interest, to which the highest texture resolution is assigned. The texture detail is then gradually reduced depending on the distance to the center of interest. The clipmap approach employs a toroidal texture packing scheme to store and allow a simple address translation of the individual texture tiles. However, the center of interest presents insurmountable problems when texture detail is required in different distant texture regions.

Other approaches such as the ones by Cline et al. [CE98] and Hüttner [Hüt98] estimated the required texture tiles and appropriate resolutions by geometric computations on the model geometry. They, however, split the original geometry to facilitate the mapping of texture tiles, making these

approaches not generally applicable. Goss et al. [GY98] proposed an extension to the standard graphics pipeline by extending the frame buffer with a per-pixel texture tile index in addition to the color and depth buffer. This makes it possible to generate feedback directly during rendering regarding what texture tiles are required and, very importantly, how many fragments are using a particular tile. This coverage information is then used to prioritize the loading of the requested tiles. A big advantage of this approach over the previous techniques is that it takes data occlusion into account. It circumvents the requirement for complex tile estimation approaches by using information available only to the renderer or graphics hardware. Lefebvre et al. [LDN04] built upon this approach and described a generalized out-of-core texture data-virtualization method. However, translucent geometries requiring multiple feedback slots per pixel are not supported by these methods. The virtualization of 3D-texture resources used for volume rendering therefore is not supported. Crassin et al. [CNLE09] extended the feedback-driven approaches to translucent volumes by employing auxiliary render targets to store spatially and temporally subsampled visibility information. They therefore generated an aggressive estimation of the actually required volume data blocks.

2.4 Summary

In this chapter fundamental concepts and rendering techniques were discussed related to the out-of-core handling and visualization of large volumetric data sets as well as height-field surface geometries.

The unified data-virtualization system proposed by this work is based on out-of-core multi-resolution representations of volume and height-field data sets, occurring in large geological models. We employ a two-level caching strategy similar to Plate et al. [PTCF02]. The working set generation in our system is based on different approaches. We support basic view-dependent criteria, but more importantly image-based criteria through feedback generated directly during rendering. We extend upon the fundamental idea introduced by Goss et al. [GY98] to issue level-of-detail requests on a per-pixel basis. While the existing approaches [LDN04, HPLVdW10] are limited to opaque geometries and therefore to single level-of-detail requests per pixel, our system supports translucent and volumetric data sets through linked lists containing a varying number level-of-detail requests. This ap-

proach inherently considers occlusions between different data sets while sub-sampling the actual screen resolution.

The presented GPU-based rendering approach for multiple large volumes builds upon our data-virtualization system. The multi-resolution volume virtualization approach used in this rendering system employs an index texture for direct access to the texture atlas cells. This way we trade increased memory requirements for reduced octree traversal computations. Binary space partitioning (BSP) volume decomposition of the bounding boxes of the cube-shaped volumes is used to identify the overlapping and non-overlapping volume regions. The resulting volume fragments are extracted from the BSP-tree in front-to-back order for rendering. This approach is similar to a technique recently presented by Lindholm et al. [LLHY09]. While they support the visualization of multi-resolution volume data sets, their approach is based on the insertion of all the volume sub-blocks in the BSP-tree resulting in a very large amount of volume fragments and rendering passes. Even though our approach is also using a BSP-tree for efficient volume-volume intersection and fragment sorting, we do not need to insert volume sub-blocks into the BSP-tree. Therefore, the BSP-tree needs to be updated only if individual volumes are moved, which is a significant advantage over techniques employing sorting on the octree brick level or using costly depth peeling procedures.

For the visualization of geological subsurface models consisting of multiple highly detailed height fields, we introduce an output-sensitive GPU ray casting-based rendering system. The access to the height fields is virtualized by our data-virtualization system, allowing us to treat the individual surfaces at different local levels of detail. The approach most closely related to our system is the one presented by Dick et al. [DKW09]. They rendered the height-field tiles corresponding to the selected quadtree cut individually in front-to-back order. To prevent fragment overdraw they used an additional rendering passes prior to the actual ray casting of each tile to mask out previously discovered height-field intersections, generating additional rendering overhead as it requires multiple render-target changes for a single tile. In contrast, our approach is able to handle entire horizon stacks composed of multiple large height-field data sets in a single rendering pass, generating virtually zero fragment overdraw. We are using a method built on the basic idea described by Policarpo et al. [PO06] to render multi-layered height-field geometries in a single rendering pass. They treated the layered height field as an atomic resource intersecting all layers in parallel during the ray

traversal. In contrast, our approach handles each horizon layer individually. This allows us to specifically handle various parts of the distinct horizons at locally different levels of detail (e.g. occluded parts on single horizons at a much lower resolution). Furthermore, we employ a minimum-maximum quadtree over the tiled horizon height fields to speed up the ray traversal [OKL06, TIS08], and use sorted intersection intervals for the individual horizons to restrict the actual intersection searches.

Our visualization system for entire geological models consisting of highly detailed stacked horizon surface geometries and massive volume data integrates the previous approaches to a combined rendering system. Through our unified data-virtualization system, we inherently deal with occlusions between different data types in a combined visualization without the application of costly occlusion culling techniques. Ultimately, this system offers high levels of flexibility to easily design custom scientific visualizations independent of the size of the underlying data sets.

Chapter 3

Data Virtualization

A CENTRAL ASPECT of this work is the handling of extremely large two and three-dimensional data primitives in the context of seismic data collections. Beyond the processing and visualization of individual parts of these collections this work is concerned with the combined management and visualization of a variety of different parts of seismic data sets. In this chapter we describe a unified data-virtualization system that is able to simultaneously handle multiple large volume and large height-field image resources with varying levels of data abstraction for the rendering algorithms presented in the subsequent chapters. In the ensuing sections we start with an overview and analysis of the problem setting and use cases for this system. We then briefly outline the complete system, before detailing design choices of individual parts of the data-virtualization system. We conclude the chapter with a discussion of practical experiences and results of a prototypical implementation of the presented system.

3.1 Problem Setting

Seismic data sets are composed of multiple different data primitives. In this thesis we concentrate on the two most important components of seismic or geological models: the seismic volumes and the layered horizon surfaces (cf. Section 1.1). In the context of seismic data, both data types use similar underlying data representations based on uniform rectangular grids: a seismic volume is a scalar field represented by a three-dimensional volume grid, whereas individual horizon surfaces are described by two-dimensional partial height fields representing scalar elevation values. Figure 3.1 shows these two distinct data types of interest.

Most seismic data sets feature a combination of interpreted horizon surfaces, seismic volumes and other derived volumetric attributes. Geological

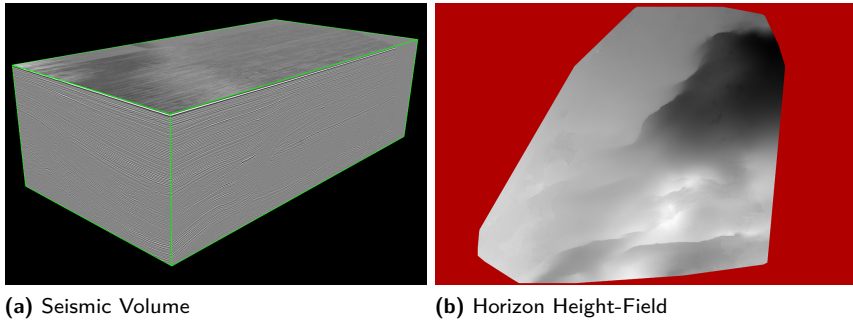


Figure 3.1: Different types of individual components of a seismic data set. Image (a) shows a seismic volume representing raw reflection amplitude values. Image (b) shows a partial height field representing a horizon surface with red color indicating unavailable data.

models typically contain multiple horizon surfaces stacked on top of each other, which are defined by individual height fields. Since the horizons are directly derived from the seismic volume, they all share a common coordinate system with the volume representing the reference frame. Through multiple partial surveys of larger subsurface regions or reacquisition of in-production oil fields, it is not uncommon that a geological model consists of a large number of volumes with different resolutions and orientations, which may be partially or even fully overlapping.

Each individual part of such a seismic data collection is potentially larger than the available graphics memory and might even exceed the size of the system memory. At no point during the run-time of a seismic visualization system can the complete data be stored *in-core*, meaning in fast system memory or graphics memory. For this reason we need to support efficient out-of-core handling of multiple data sets of different underlying data types, which means that only a smaller working set of parts of an entire data set is held resident in host and graphics memory. With the very specific application of this system to seismic data visualization, a set of special requirements apply extending beyond traditional out-of-core visualization approaches.

3.1.1 Problem Analysis

The thesis at hand pursues the goal of providing a scalable out-of-core data management and virtualization system for the specific application of scientific visualization of seismic data sets. The main purpose of an out-of-core data management system for the visualization of large data sets is to overcome the limitations of the employed graphics hardware in terms of restricted memory size foremost, as well as restricted memory bandwidth and processing resources.

The most fundamental requirement to our data-virtualization system is the ability to handle multiple large volume grids and multiple large height-field images simultaneously. Regarding an exemplary visualization of a seismic model as shown in Figure 3.2, we can deduce more specific observations: while horizons are typically displayed in full, the volumes are visualized mostly only in part through the use of cubical volume lenses. Some model components are displayed translucently and large parts of the individual data set components are invisible due to mutual occlusions and data clipping (e. g. by volume lenses or the viewing frustum). Furthermore, as illustrated by the bottommost horizon, a very desirable feature in seismic visualizations is to map volumetric attributes to individual surfaces through a direct volume lookup at each visible surface location. Based on this example and observations of further visualization configurations in the oil and gas domain, the following summarizes the most important observations:

- ▶ Seismic data sets are comprised of multiple data primitives, with the most important primitives being represented by two-dimensional horizon height-fields and three-dimensional volume grids. Each primitive in the data sets potentially exceeds the available in-core memory resources.
- ▶ Visualizations need to deal with multiple, potentially spatially non-aligned volumetric data sets, depicting different seismic surveys as well as additionally derived data attributes.
- ▶ Visualizations contain multiple layered horizon height-fields. These stacked elevation maps are directly derived from large volumetric seismic surveys.
- ▶ In a combined visualization, individual data set components may be displayed translucently. This is not limited to a direct volume rendering of a seismic. Depending on the use case, the volume lenses can also be

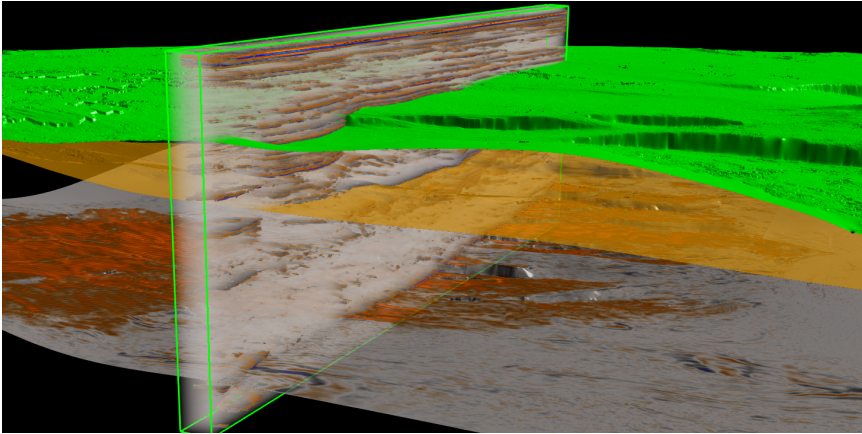


Figure 3.2: A typical visualization of a seismic model portraying the combined rendering of partially translucent horizon surfaces and a direct volume rendering of a portion of the seismic volume through a volume lens.

rendered completely opaque as a reference to individual horizon surfaces which are rendered translucently.

- ▶ The irregular geometry and the potentially large number of horizon surfaces introduce a high depth complexity into the visualizations.
- ▶ Seismic visualizations show different forms of data occlusion: self-occlusion of individual horizon surfaces and self-occlusions of the volumes; and occlusion among different horizons as well as inter-data occlusions of volume and horizon primitives.
- ▶ Volume data is potentially mapped to quite arbitrary geometries, as demonstrated by the volume attribute displayed on the bottommost horizon.

These observations regarding the visualization of seismic models present the central problem setting of the specific out-of-core data management system presented in this chapter. While parts of this system are generalizable to other problem domains such as medial visualizations or terrain rendering, the presented solutions are aligned with the special requirements of scientific visualizations in the context of seismic models.

3.1.2 System Requirements

An out-of-core data management system for the purpose of handling large seismic models in the previously illustrated application context must consider the implications of the presented observations. Based on these observations, we derive the following key requirements for our system:

- ▶ **Scalability:** The system must scale in regard to individual data set sizes as well as the number of concurrently handled data sets.
- ▶ **Multi-Resolution Data Representations:** The system must use multi-resolution data representations to facilitate flexible level-of-detail and out-of-core handling of the individual data set components.
- ▶ **Flexible data virtualization:** Depending on the use case, the system must offer fully transparent data virtualization but simultaneously offer the ability to access the underlying specific data representations to facilitate exploiting their structure in order to potentially accelerate the rendering algorithms.
- ▶ **Occlusion-aware data selection:** Through the high depth complexity of seismic visualizations, considering data visibility is crucial for selecting appropriate working sets that exploit the available system resources most efficiently.
- ▶ **Unified out-of-core data management:** All data primitives must be first class data types. The different data types must be handled in an integral manner without special treatment of individual primitives. Privileged treatments result in resource starvation of individual data types, potentially causing visual disparities among distinct data entities.
- ▶ **Shared and non-concurrent resource management:** System resources such as host and graphics memory but also bandwidth resources must be shared and balanced for all data types and data primitives handled by the system. Resources freed up by one primitive must be made available to other primitives and, conversely, resources vital to a particular primitive must be withdrawn from others after taking their actual demands into account.
- ▶ **Non-blocking, demand-driven data caching:** The system determines data blocks required for the actual working sets. These must be moved from external storage to internal caches in local memory without blocking the operation of the driving rendering approaches.

Existing out-of-core rendering approaches are specialized for individual data primitives, such as large volumetric data sets or large height-field surfaces (cf. Section 2.3.3). The requirements of our system are more diverse. Foremost, it needs to handle two different data types simultaneously and needs to efficiently scale with regard to the data set sizes as well as the number of concurrently handled data sets.

The system must, in a general sense, implement a data-virtualization approach. The term *virtualization* refers to the abstraction of logical data resources from their actual physical characteristics. They effectively hide the complexities of out-of-core and multi-resolution data representations, such as the employed internal data structures and memory layouts as well as the currently available data, to allow to treat the data as if it could be handled entirely as if in-core (cf. Section 2.3). With our system, we must support varying levels of data abstraction for different usage scenarios. For example, mapping a volume attribute to a horizon surface is most effectively handled fully transparent to the underlying data representations, while volume ray casting algorithms can use the multi-resolution hierarchy to accelerate ray traversal and optimize volume sampling.

A virtualization system for interactive scientific visualizations is different than a virtual memory management system in today's operating systems. Upon detecting missing entries in a data cache, an interactive visualization must not stall and wait for the data to arrive from secondary storage. An interactive visualization system requires a representation of the data at any point during run-time. Multi-resolution approaches allow for representation of the data at locally varying levels of detail. Therefore, a data-virtualization system for real-time rendering purposes can fall back to coarse data resolutions while required data is progressively paged in.

3.2 System Overview

In this section we will give a brief overview of the basic architecture of our data-virtualization system and describe the relationships of the most important components. The following sections of this chapter will then present the individual system components in more detail.

Our system is designed for the efficient out-of-core management of large geological models consisting of multiple horizon surfaces and seismic volumes. Fundamentally, the underlying uniform grid data primitives are represented

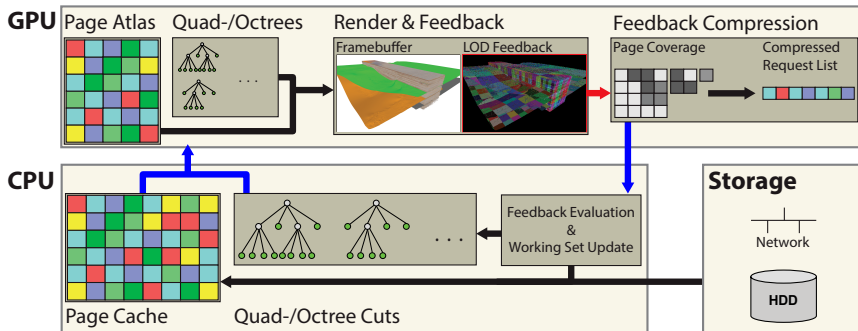


Figure 3.3: This figure shows an overview of the complete out-of-core data virtualization and rendering system.

as textures on the GPU. The height fields describing the horizon surfaces are managed as 2D single-channel texture images and the volumes are managed as 3D-texture resources. While these data types are of a scalar nature, the basic design of this system does not generally distinguish between multi-channel and single-channel data resources. Figure 3.3 illustrates the basic system architecture. The out-of-core data management system consists of three main parts: the page cache, the page atlas and the level-of-detail feedback mechanism. The page cache is managed in main memory and used for fetching data from external storage. The page atlas in GPU memory stores the actual working sets used for rendering. The update of the working sets is driven by level-of-detail feedback information gathered directly during rendering.

Data Representation

The system employs hierarchical multi-resolution data representations for the level-of-detail and out-of-core handling of the individual height-field and volume data sets, based respectively on quadtree and octree data structures. While the leaf nodes of these hierarchies represent the highest resolution data, the inner nodes hold low-pass filtered versions of the source data (cf. Section 2.3.3). All nodes in the quadtree and octree hierarchies are represented by data blocks of a fixed size. These blocks act as the basic paging unit throughout our memory management system and we therefore

refer to them more generally as *data pages*, independent of their type. The multi-resolution hierarchies of the individual data sets are generated in a pre-processing stage. After the pre-processing of the data sets is finished, we additionally support the compression of the data pages for faster page transfers during run-time. The compressed pages are finally stored in out-of-core page data pools located in secondary storage, such as a hard drive or network share.

Data Management

The resource management of our out-of-core data management system is based on the idea of a two-level cache hierarchy. This hierarchy is defined by two large resources: the page cache in system memory acting as a second-level cache for page data loaded from external sources and the page atlas in graphics memory as the first-level cache for the working set of page data used during rendering. Both cache stages are defined as global resources to all data sets of a common type which are handled by the system. This enables us to efficiently balance the individual memory requirements of distinct data sets against global memory constraints. While we employ a LRU - least recently used - strategy to replace pages in the page cache, the paging of pages in the page atlas is controlled by the update method of the individual representations of the multiple handled data sets.

The page atlas is conceptually implemented as a single large texture in GPU memory, making it possible to leverage the fast bilinear and trilinear filtering capabilities of the GPU. Through the technical limitation that texture resources on the GPU are strictly bound to a certain texture type and format, we have to handle separate page-atlas textures on the GPU for height-field and volume data. However, these texture resources are then global to all data primitives conforming to a certain type. Therefore, memory resources are only shared among data primitives of the same type, but the system still considers shared constraints such as bandwidth limits globally for all data types.

Working Set Generation

For each individual data set, we compute and continuously update a cut from its multi-resolution hierarchy. The system follows a modular design to allow different strategies for the prioritization of data pages to steer the cut

update method. We support view-dependent updates, considering object space distance metrics, and an advanced feedback mechanism, gathering information about required data pages directly during rendering. Through the latter approach we are able to inherently deal with each possible form of data occlusion in seismic models without the application of complex visibility detection algorithms. Our approach directly emits data-page requests on a per-fragment level basis during the rendering process, depending on the actually required data resolutions. This way we achieve a demand-driven caching strategy considering actually visible data pages.

All nodes that are part of a multi-resolution hierarchy cut define the actual working set of height-field and volume pages that are stored in the atlas textures in GPU-bound memory. The cut updates are incrementally performed using a greedy-style split-and-collapse algorithm, which exploits frame-to-frame coherence. The update algorithm considers the fixed texture memory budget defined by the page atlas textures as well as a bandwidth budget for the upload of new data pages to the GPU. During the update operation, only data currently residing in the main memory-page cache is considered and unavailable pages are requested to be asynchronously loaded and decoded from the out-of-core page pool. This approach prevents stalling of the update and rendering process due to slow transfers from external page sources.

A pre-fetching mechanism is employed when the actual required working sets for an adequate visual representation of the data sets are not fully utilizing the available host and graphics memory resources. In such cases this mechanism extrapolates the available data visibility information to predict potentially required data pages of future rendering frames. These data pages are then scheduled to be loaded ahead of time to increase the cache efficiency of the system in order to allow smooth roaming through large data sets.

Data Virtualization

During rendering, the actual data sets are generally accessed through two resources on the GPU: a large page-atlas texture of a fixed size containing all pages of the current working sets of all data sets of a common type (e.g. height-field image, volume); and a set of small auxiliary textures containing indirection information about how to locate data pages and translate virtual texture coordinates to physical sample coordinates in the page atlas. At this

point the system is again designed in a modular fashion to facilitate different application scenarios. We support two approaches for the encoding of the indirection information: a page table per data set or a compact serialization of the hierarchy cut associated with a data set. The page table makes it possible to retrieve the indirection information directly depending on the virtual sampling location and therefore introduces a constant sampling and run-time overhead. In contrast, the sparse serialization scheme represents a much more compact encoding of the hierarchy cut, but introduces logarithmic traversal cost for a single data lookup.

3.3 Multi-Resolution Data

The most essential choice in the design of an out-of-core data management system for large texture resources is their internal data representation. This choice fundamentally influences large parts of the system such as memory management and virtualization sub-systems. Through the close relation of the two data types this work is focusing on, we chose closely related multi-resolution representations for 2D and 3D-texture resources. This allows us to unify most of the data management system. In the following sections of this chapter, we describe our design choices related to the multi-resolution representations of height-field and volume data sets.

3.3.1 Data Representation

The basic approach for the management and working set definition of individual large data sets, represented by uniform grids, is subdividing the source data into fixed size blocks. Based on this data blocking, two choices exist to build a multi-resolution representation of the source data set: hierarchical blocking and flat blocking schemes (cf. Section 2.3.3).

Hierarchical blocking schemes respectively build an octree or quadtree hierarchy by recursively filtering adjacent blocks bottom-up from the initial blocked data set until the root node represents the entire data set at the coarsest resolution. Figure 3.4 illustrates the creation of a multi-resolution of a two-dimensional quadtree. An important characteristic of such hierarchical blocking schemes is that the number of data blocks is reduced with each level in the hierarchy, while all nodes represent a data block of the same fixed resolution. Therefore, all nodes in such hierarchies are represented

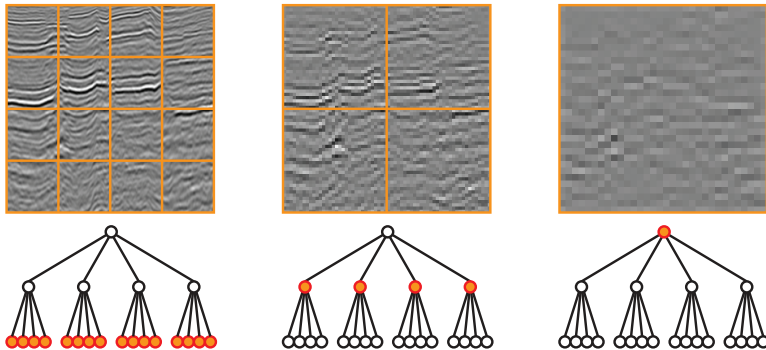


Figure 3.4: Creation of a hierarchical multi-resolution data representation. The original data set is subdivided into fixed size blocks, represented by the leaf nodes of the hierarchy. Coarser data resolutions are represented by inner nodes and are generated by successively combining and filtering adjacent blocks on finer levels.

by blocks of the same size, as the spatial extent of the blocks is increased. Hierarchical blocking schemes are traditionally used in the visualization of large volume data sets [LHJ99, WWH⁺00, BNS01, PTCF02, GWGS02]. In contrast, flat blocking schemes keep the spatial extent of the original data blocks constant while successively reducing the resolution of the individual blocks. The amount of data blocks is kept constant and each block is thereby represented by a mipmap hierarchy with a fixed number of resolution levels. With defined constraints on the maximum data-page size, the actual number of data blocks can increase greatly for large volume data sets. For this reason, flat blocking schemes are generally applied only to moderately sized volumes [Lju06, BHMFO8].

Choice of Multi-Resolution Representation

We choose hierarchical multi-resolution approaches based on octree and quadtree structures for the representation of volume and height-field data in our system. Hierarchical approaches offer particular advantages over flat blocking methods when working with extremely large data sets. As the data blocks of the chosen blocking scheme act as the basic paging unit throughout our system, the fixed page sizes of the hierarchical approach

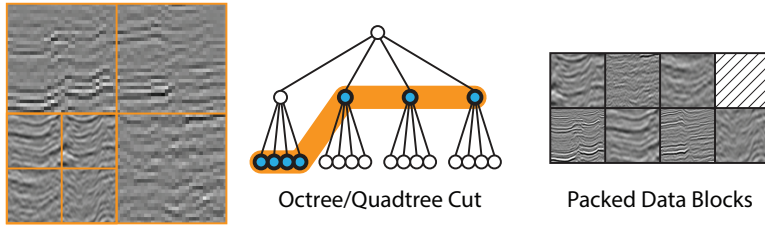


Figure 3.5: These images illustrate the packing of data blocks using a hierarchical multi-resolution data representation based on fixed block sizes.

facilitate straightforward cache and memory management. Moreover, the rigid spatial subdivision and therefore fixed amount of data blocks making up a data set associated with flat blocking methods would create large memory and computational overhead for larger data sets.

Considering a fixed memory size for the storage of data pages in either CPU or GPU bound memory, the fixed page size of the hierarchical approach allows for a rigid subdivision of the memory block to form a page pool of a defined page capacity. The resulting fixed amount of data pages represents a constant page budget for the management of the working sets of the data sets. These working sets stored in the page pools are defined by the cuts from the multi-resolution hierarchies. Therefore, the size of the hierarchy-cut representations is a function of the memory available to the data management system, which is typically multiple orders of magnitude smaller than the actual data set sizes. Based on the fixed page size, page insertion and replacement operations can be handled in a straightforward manner without introducing any memory fragmentation.

On the other hand, flat blocking approaches inherently employ a fixed spatial subdivision of the original data sets with varying data block sizes. They therefore offer no fixed page size and no fixed page pool capacities. Considering a maximum size for a data page, the actual size of the data subdivision is a function of the actual data set size. Hence, the data structures describing these uniform grids are directly dependent on the data sets, whereas hierarchical approaches only depend on the memory size handled by the system. Therefore, due to the fixed spatial subdivision of the flat blocking approaches, the number of required data pages is constant and the size of the data pages must be adapted to steer the size of the working

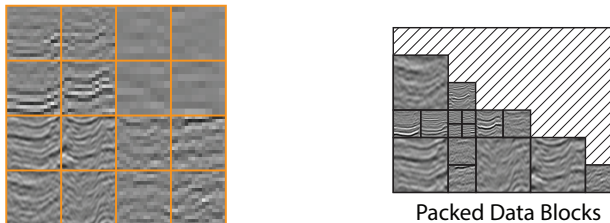


Figure 3.6: These images illustrate the packing of data blocks using a flat multi-resolution data representation with inherently varying block sizes.

sets. Consequently, additional memory management overhead is introduced due to memory fragmentation problems. The management of a page pool requires the application of flexible bin-packing algorithms to minimize the memory fragmentation issues while dynamically removing and inserting data pages during system run-time [Jyl10]. Figures 3.5 and 3.6 highlight the difference in data packing for the two different blocking schemes.

Based on these observations, we conclude that hierarchical multi-resolution approaches are much more suitable for the out-of-core management of large regular grid data sets such as large volume and image texture resources. Their compact data representation allows for a much better adaptation of the data structure sizes to the actual memory constraints. Furthermore, hierarchical data structures are very well suited for adaptive rendering approaches. The alignment of the spatial extents of a data block to the actually represented data resolutions allows for efficient volume ray casting implementations. For example, a ray traversing a volume based on an octree data structure intersects considerably less nodes describing data bricks than using a uniform grid data structure of flat blocking approaches, thereby reducing the data structure traversal overhead. This overhead is again a function of the memory available to the rendering system and not dependent on the data set size.

3.3.2 Data Preparation

The multi-resolution quadtrees and octrees representing the height-field and volume resources in our data management system are generated in a pre-process. Based on an initial blocking of the source data sets, the

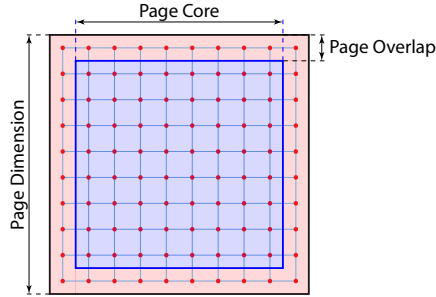


Figure 3.7: This image illustrates the basic parameters of a data page. The page core represents the actual data samples associated with the particular page. The page overlap represents the shared boundary between adjacent pages.

hierarchies are created based on box filters calculating the arithmetic mean of respectively four or eight data samples to generate an associated sample on the next coarser level. We deal with non-power-of-two data sets by virtually padding the data set to the next full power of two dimension and introducing empty nodes into the hierarchy. Therefore, only a small overhead is introduced for data blocks on the edges of the data sets, which only partially fill a data block.

In order to allow correct data interpolation at block boundaries, we explicitly generate a border around each block of shared data samples from adjacent data blocks (cf. Section 2.3.3). The size of the shared border can be adapted depending on particular application scenarios. Regular interpolations require just a one sample wide border, while gradient calculations generally require a two sample wide border between adjacent blocks.

In fact, the shared block borders are not added to the data block, they reduce the amount of usable data per block as the page dimensions are kept stable. We refer to these data blocks generally as *data pages* in the context of our out-of-core data management and data-paging system. The actually usable data block associated with each data page is called the *page core* and the overlapping shared boundary is referred to as the *page border*. Figure 3.7 illustrates the most important properties of a data page in our system. Through this view on the relation of page core and border, we are able to handle different data sets with varying overlap dimensions

Dimension	Overlap	Volume				
		16^3	32^3	64^3	128^3	256^3
Size		4 KiB	32 KiB	256 KiB	2 MiB	16 MiB
Overhead	1 Voxel	33.0%	17.4%	9.9%	4.6%	2.3%
	2 Voxel	67.8%	33.0%	17.4%	9.9%	4.6%

Table 3.1: Volume page memory overhead for one and two voxel wide page borders. The volume page sizes are calculated based on typical 8 bit fixed-point voxel precision.

equally by keeping the page dimensions constant. Through the replication of shared data samples for neighboring bricks, a high degree of redundancy is generated in the multi-resolution representations. Depending on the chosen page dimensions, this redundancy introduces different levels of memory overhead. Tables 3.1 and 3.2 illustrate the incurred costs for different volume brick and height-field tile dimensions for a one and two sample wide border. Approaches that forgo the explicit duplication of samples at block boundaries have been proposed in the recent past [LLY06, BHMF08]. They manually gather and interpolate data samples at the boundaries of individual blocks to generate correct transitions. Such approaches, however, introduce large run-time overhead to the determination of the required sample locations and the manual interpolation. We chose to avoid such run-time overhead at the cost of higher redundancy and therefore memory overhead.

Dimension	Overlap	Height Field		
		128^2	256^2	512^2
Size		32 KiB	128 KiB	512 KiB
Overhead	1 Texel	3.1%	1.6%	0.8%
	2 Texel	6.2%	3.1%	1.6%

Table 3.2: Height-field image page memory overhead for one and two texel wide page borders. The Height-field page sizes are calculated based on typical 16 bit fixed-point texel precision.

Apparently, the use of larger data pages will minimize the memory overhead inherent to duplicated data samples at page boundaries. However, we have to account for a wider range of factors. Transfer rates and access latencies of secondary storage depend on the size of consecutively read data blocks. The update of texture resources, especially partial updates as required by our texture atlas approach, are heavily affected by the choice of data-page sizes. Coincidentally, these sizes also influence the granularity of hierarchical rendering algorithms based on the multi-resolution structures. Smaller data pages will result in more traversal overhead, larger data structures and provoke increased texture cache thrashing. We discuss practical experiences with different page sizes in Section 3.8.1 and detail our choices for different applications.

3.3.3 Data Indexing and Storage

After establishing the choice of hierarchical multi-resolution data representations as the basis for our system design, we now illustrate the indexing and storage of the multi-resolution hierarchies.

Hierarchy Indexing

In order to uniquely identify nodes in individual multi-resolution hierarchies, we employ continuous indexing methods based on a z-order curve for the quadtrees and octrees. The z-order curve, or Morton curve [Mor66], offers

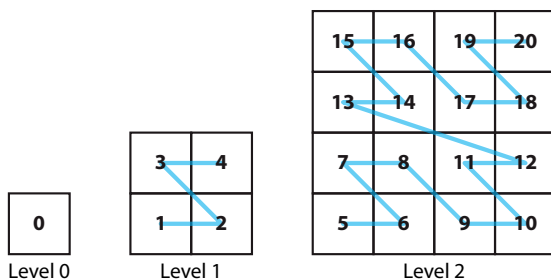


Figure 3.8: Continuous indexing of a quadtree hierarchy by using a z-order space filling curve. This image shows three iterations of the curve assigning unique identifiers to each node.

several advantages we exploit throughout our system, such as for the definition of a cache-coherent file layout or the unique identification of hierarchy nodes. The z-order curve resembles the order of nodes resulting from a breadth-first traversal of a quadtree or octree (cf. Figure 3.8). This property represents a major advantage of using the z-order curve to serialize the tree hierarchies. It preserves the locality of the tree nodes: neighboring nodes in the hierarchy are represented by a close range of index values. Therefore, this indexing approach facilitates an inherent cache-coherent layout for the external storage of the pre-processed multi-resolution hierarchies. Furthermore, it facilitates straightforward index calculations of parent and children nodes in the hierarchies. The actual index of a node is calculated by interleaving the binary representations of the coordinate values of its position in the tree hierarchy \mathbf{p} added to the total number of nodes on the levels above the nodes hierarchy level l :

$$\begin{aligned}
 \text{Quadtree: } N(\mathbf{p}, l) &= \quad \text{bin_interleave}(\mathbf{p}_x, \mathbf{p}_y) \\
 &\quad + \text{total_node_count}_2(l-1) \\
 \text{Octree: } N(\mathbf{p}, l) &= \quad \text{bin_interleave}(\mathbf{p}_x, \mathbf{p}_y, \mathbf{p}_z) \\
 &\quad + \text{total_node_count}_3(l-1)
 \end{aligned} \tag{3.1}$$

Therefore, starting from a given index we can calculate child and parent indices as well as retrieve the actual node position and level in the tree hierarchy. The actual calculations can be expressed as a series of low-level bit operations on the binary integer representations of the node positions and indices and are therefore very quickly executed.

Throughout the out-of-core memory management system, the node indices are employed as unified *page ids*. These page ids are equally used for quadtree as well as octree nodes, with no special distinction at this abstraction level. As the indices representing the page ids are just integer numbers continuously running from 0 upwards, we distinguish between pages of different data sets by adding a so called *instance id* to each identifier. Through this id we can uniquely attribute each data page to its associated data set.

Because we are using the page ids as the main instrument of storing and communicating page identities, we chose a very compact binary representation of the combination of instance and page ids. They are implemented as a 32 bit-wide binary bit-field. Considering a partitioning of the page id of 8 bit allocated to the instance id and 24 bit for the z-order curve index,



Figure 3.9: The basic file layout for the storage of individual multi-resolution hierarchies. The index section stores offsets for all non-empty data pages in the data section.

we can support 256 data set instances of up to 16.8 million unique nodes. As a result the representation of octrees would be limited to hierarchies with a maximum depth of 7, as an octree of depth 8 already requires 19.2 million unique indices. We therefore designed the bit-field partitioning as a compile-time variable allocation of bits to the instance id and page index to allow for balancing of supported tree sizes and amount of simultaneously distinguishable data sets.

Data Storage

All individual pre-processed multi-resolution hierarchies are stored on secondary and external storage locations (e. g. hard drive, network storage). Figure 3.9 illustrates the file layout we designed for our out-of-core data management system. It contains three sections: the file header, the index section and the page data section. The file header describes the stored data set and holds information about the data format, the chosen page dimensions and page overlap. The index section stores offset information to the page data in the data section for all non-empty hierarchy nodes. This index section is accessed by the z-order page-index and results in the file offset of the associated page data. Invalid offset values indicate empty nodes for error assertions in case of erroneous data requests of the system.

In order to save storage space as well as data transfer bandwidth from the secondary storage to the host system, we support the storage of compressed data pages in our file format. A number of different compression methods are available; however, lossless compression methods are favored as this system is primarily applied to scientific visualizations, where lossy compression is not desired. This requirement can be loosened by applying different compression methods to data pages representing the original data and data pages representing already lossy filtered, coarser levels of detail in the multi-resolution hierarchies. However, independent of the chosen compression

method, the sizes of the stored and tightly packed data pages vary. In the event that a compressed file format is specified in the file header, the index section is used to calculate the page data sizes as a difference of the offsets of the particular node's own index and the index of the successive node according to the z-order curve index. A difference of less than 1 thereby again indicates empty nodes.

Through the employed z-order for storing the data pages in the file structure, we make use of its locality properties. Adjacent nodes in the multi-resolution hierarchies are stored close to each other. Therefore, when accessing a certain node it is very likely that in the near future neighboring nodes will also be accessed. Our file structure represents a cache-coherent data layout that makes it possible to consecutively read data pages and facilitates efficient data pre-fetching from secondary storage.

3.4 Texture Virtualization

The key to decoupling multi-resolution data representations and rendering algorithms is an efficient virtualization of the multi-resolution texture hierarchy. Texture virtualization refers to the abstraction of a logical texture resource from the underlying data structures, effectively hiding the physical characteristics of the chosen memory layouts.

Multi-resolution hierarchies are employed for the visualization of large volume, terrain and image data sets throughout the past decades in various application areas and rendering systems. Most traditional GPU-based implementations of rendering algorithms for such blocked data representations are implemented through multi-pass methods due to limitations of the current hardware generations available at that time. However, besides the usual increase of processing performance, GPUs especially showed a dramatic evolution toward very flexible and widely programmable co-processors to the host systems CPU. Nevertheless, they represent a separate device with particular properties and its own memory sub-system connected to the host through a system bus. Managing and accessing complex data structures on the GPUs still differs from CPU-based implementations. In the following sections, we illustrate the most important data structures for efficient representation of the multi-resolution-based working sets on the GPU and detail our approach of transparent texture data virtualization through a global virtual texture management system.

3.4.1 The Virtual Texture System

The purpose of the *virtual texture system* is the management of large texture resources employing multi-resolution data representations and the abstraction of the chosen memory layouts from developers or visualization algorithms. As we want to support multiple data sets of different types simultaneously without concurring access to system resources, such as memory or data transport bandwidth, we chose to design a global management system which considers memory requirements of all data sets and shares global system resources among all data sets.

Page Atlas Texture

Modern GPUs are limited in the amount of individual texture resources they can address from a single shader program¹. With memory sizes of up to 6 GiB on the current generation of GPUs and usual data pages sizes of 256 KiB to 2 MiB (cf. Section 3.3.1), a working set stored on the GPU can hold thousands of pages simultaneously. Hence, such working sets are not trivially manageable due to the current texture resource limitation².

However, it is highly desirable to allow rendering algorithms access to the entire working set of data pages constituting an individual large data set represented by a texture resource (e.g. single-pass volume ray casting methods). Our solution resolves this problem by employing a single shared atlas texture, storing all data pages of all working sets combined [Nvi04]. The pages are tightly packed into the atlas texture as illustrated in Figure 3.10 for two seismic volume data sets. This way we can take advantage of powerful GPU capabilities such as inherent 2D and 3D-texture addressing, fast texture interpolation and optimized texture caches. Similar approaches were proposed coincidentally to the research work for this thesis by Gobbetti et al. [GMG08] and Crassin et al. [CNLE09] in the context of GPU-based large volume rendering. They, however, purely focus on individual data sets of a single particular type.

¹The number of *texture image units* to access individual texture resources from a single shader program is limited to 16 (NVIDIA G8x-GT2xx series) or 32 (NVIDIA GF1xx-GK1xx series).

²The current generation of NVIDIA GPUs from the GK1xx series lifts the texture binding limit through the proprietary bindless texture functionality (`NV_bindless_texture`).

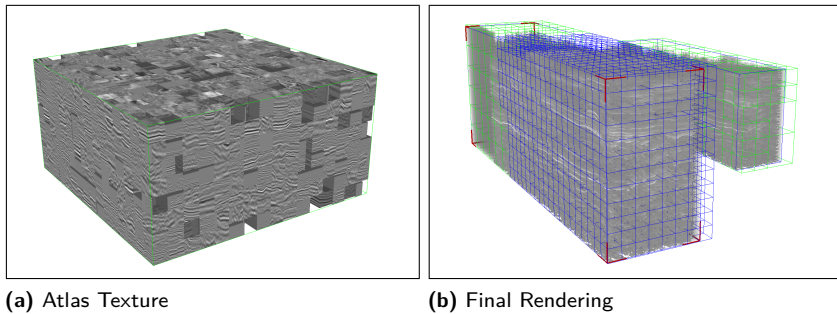


Figure 3.10: Volume visualization using virtualized multi-resolution textures. (a) shows the atlas texture containing volume brick data of two volumes. (b) shows the final rendering based on volume ray casting.

Typed Virtual Texture Contexts

An obstacle of a truly shared texture memory resource on the GPU, global to all data sets, is the fixed typing of texture resources. A texture is defined by a combination of a texture type and format. The texture type (2D or 3D-texture) defines the memory layout and the format defines the data type represented by the texel values, for instance scalar 8 bit fixed-point voxels or 32 bit floating point values. A solution to this problem would be the storage of mixed type page data in linear GPU memory, abandoning every advantage of dedicated texture sampling hardware and implementing custom data sampling and filtering algorithms. Such an approach is not feasible on current hardware generations as the processing and data access overhead does not allow comparable performance to the use of dedicated texture memory. We therefore opted to use multiple atlas textures appointed to individual data types in our system. This means that we have to manage at least two large texture resources on the GPU: a 2D-texture atlas for the management of height-field pages and a 3D-texture atlas for the storage of volume pages.

In order to scale to more than these two types of data resources, we abstracted all specifically type related resources into so called *virtual texture contexts* or, for short, a *vtexture_context*. These contexts are defined by the type, the format and the page dimensions used for particular data sets.

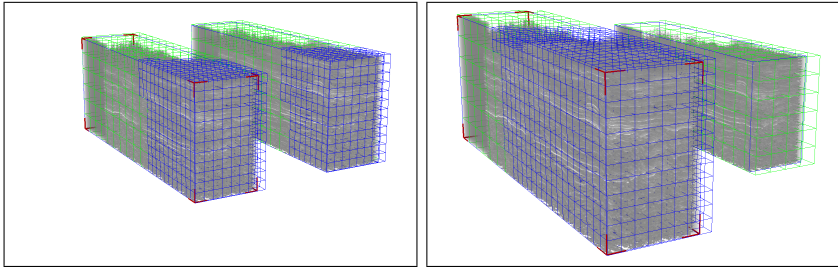


Figure 3.11: Texture resource distribution between two seismic volume data sets during a zoom-in operation. The size of the bricks in the multi-resolution octree cuts, shown as wire frame overlays, show the local volume resolution. Blue boxes represent the highest volume resolution while green boxes indicate lower resolutions.

They store a large atlas texture corresponding to the specific type and format as well as a memory pool in the host system acting as the second level cache for this context. These resources are global, shared resources for all data sets of a common type that need to be handled at a time. In contrast to individual, non-shared resources attached to each data set, this allows us to balance the memory requirements between different data sets. If, for example, volumes are moved out of the viewing frustum or are less prominent in the current scene, the unused resources can be easily shifted to other volumes without costly reallocation operations in the host or graphics memory (cf. figure 3.11).

The management of the individual virtual texture contexts is currently implemented independently in our system. The number of contexts and their properties including the actually allocated memory sizes are defined by the developer of the particular application. Therefore no memory allocation balancing between different data types is supported. However, an extension allowing dynamic and transparent reallocation of the texture and host memory resources can soften this constraint.

Current generations of GPUs support a feature called texture arrays, which we exploit for the extension of a 2D-texture atlas. Using this extension, we can address multiple identically typed and sized 2D-textures through a single texture resource. This is important because texture-size restrictions

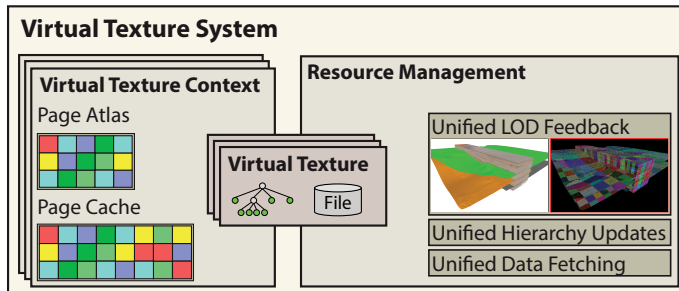


Figure 3.12: This figure shows an overview of the dependencies of the components of our virtual texture management system. Virtual texture contexts hold a set of virtual textures and the shared memory resources specific to a particular data type. The system globally manages the memory resources of all handled virtual textures.

of current GPUs prevent us from taking advantage of most of the available memory. For example, the current maximum size restriction of 16384×16384 of 2D-textures applied to a 16 bit texture format allows us to access only 512 MiB of potentially available 2 to 6 GiB of GPU memory. By applying texture arrays this limitation can be circumvented.

Generalized Virtual Texture System

Virtual texture contexts purely act as containers for the specifically context-type dependent resources. The management of the resources is handled solely by the unified *virtual texture system*. This leads to the system abstraction illustrated in Figure 3.12. The virtual texture system manages a set of virtual texture contexts. These contexts themselves hold a set of virtual textures and their shared memory resources specific to a particular data type. The virtual texture system then manages the update of the individual multi-resolution hierarchy cuts based on a unified level-of-detail feedback mechanism and offers global data fetching facilities. This way all memory transfers, such as reading data from external storage, writing data pages into particular page-cache memory or uploading data pages to the GPU, are handled without any concurrent access conflicts of specific global memory or bandwidth resources.

3.4.2 Texture Abstraction

The design of our texture virtualization system provides access to the working sets constituting multi-resolution data representations of large data resources through single texture resources - the page atlas textures. These textures contain all data pages associated with the current hierarchy cuts, which represent large volume or height-field primitives.

Regular texture resources are generally accessed through a set of texture coordinates that are used to directly retrieve a filtered data sample. With virtual textures, a mechanism is required for the translation of *virtual texture coordinates* into a set of texture coordinates addressing the associated physical sampling location inside the associated data page in the page atlas texture. Figure 3.13 illustrates this procedure. For the production of the indirection information, our system offers two possibilities with different tradeoffs:

- ▶ Run-time overhead to generate coordinate translation information.
- ▶ Size of the involved data structures.
- ▶ Efficiency of data structure updates.

In the following parts of this section, we detail the virtual coordinate translation calculations and describe a page-index texture and a tree-serialization scheme encoding the indirection information. While the page-index texture approach facilitates fast and direct access to the stored information, it involves a larger storage and update overhead compared to the serialization scheme. The serialization offers compact storage and fast data structure updates, but introduces a run-time overhead for the access of the indirection information.

Virtual Texture Coordinate Translation

Virtual texture coordinates in two or three-dimensional spaces are vectors with each component defined in the interval $[0.0, 1.0]$. These virtual coordinates represent the sampling location in the original data set. However, this data set is stored in a scrambled and blocked form in the physical page atlas texture. The atlas texture itself is also accessed through texture coordinates in the range $[0.0, 1.0]$. By providing the indirection information about the location and size of the physical data page in atlas texture, the virtual texture coordinates \mathbf{c}_{virt} are translated into physical texture coordinates

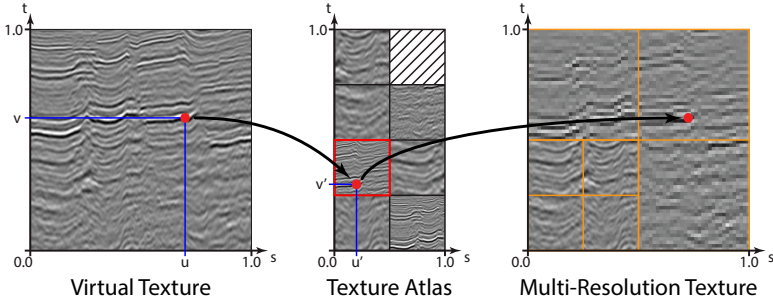


Figure 3.13: These images illustrate the procedure of a virtual texture lookup. The virtual texture coordinates (u, v) are translated into physical sampling coordinates (u', v') of the associated data page in the texture atlas resulting in the final filtered data sample that is used for the multi-resolution data representation.

\mathbf{c}_{phys} in the atlas texture. This operation is expressed as a basic bias and offset operation:

$$\mathbf{c}_{phys} = \mathbf{p}_{bias} + \mathbf{p}_{offset}(\mathbf{c}_{virt}) \quad (3.2)$$

The page bias \mathbf{p}_{bias} is directly provided through the indirection information, which is represented by a $(\mathbf{p}_x, \mathbf{p}_y, \mathbf{p}_z, l)$ tuple storing the page atlas position of the page \mathbf{p} , represented as integer values denoting the index of the page, and the level l of the associated node in multi-resolution hierarchy. The actual bias in the texture coordinate space of the atlas texture is calculated by the component-wise division of atlas page position with the atlas capacity \mathbf{acap} expressed in the number of stored pages along each axis:

$$\mathbf{p}_{bias} = \frac{\mathbf{p}_{x,y,z}}{\mathbf{acap}_{x,y,z}} \quad (3.3)$$

The page offset \mathbf{p}_{offset} then is calculated based on the virtual texture coordinate and the number of pages on the current hierarchy level as follows:

$$\mathbf{p}_{offset}(\mathbf{c}_{virt}) = \frac{fract(\mathbf{c}_{virt} \cdot 2^l)}{\mathbf{acap}_{x,y,z}} \quad (3.4)$$

The *fract* operation, returning the fractional parts of its input, results in normalized coordinates in the coordinate space of the actual data page.

Through the component-wise division of this page coordinate with the atlas capacity **acap**, we obtain the page offset in the atlas texture coordinate space. The resulting physical texture coordinate then is expressed by:

$$\mathbf{c}_{phys} = \frac{\text{fract}(\mathbf{c}_{virt} \cdot 2^l) + \mathbf{p}_{x,y,z}}{\mathbf{acap}_{x,y,z}} \quad (3.5)$$

However, this computation does not consider the borders inherent to each tile. Therefore, the page offset requires an additional offset and scaling calculation using the page core scale \mathbf{p}_{c_scale} and the page border offset \mathbf{p}_{b_offset} before calculating the physical texture coordinate:

$$\mathbf{p}_{offset_core}(\mathbf{c}_{virt}) = \mathbf{p}_{offset}(\mathbf{c}_{virt}) \cdot \mathbf{p}_{c_scale} + \mathbf{p}_{b_offset} \quad (3.6)$$

In summary, the translation of virtual texture coordinates into physical texture coordinates into the texture atlas is a straightforward operation. The computation is based on a variable indirection vector, depending on the virtual sampling location, and run-time constants describing the atlas texture and the page parameters.

Page Index Texture

The first approach for the storage of the indirection information of the current tree cuts is the use of so called *page-index textures*. These are small 2D and 3D-textures containing the indirection information tuples required for the virtual texture coordinate translation. The size of these page-index textures is defined by the size of the associated quadtree or octree, which means that a texel in the index texture stands for a leaf node in the particular hierarchy. The indirection information $(\mathbf{p}_x, \mathbf{p}_y, \mathbf{p}_z, l)$ required for the virtual texture coordinate translation is directly stored in the texels representing the particular leaf node. A tree cut is represented by filling the areas covered by the coarser nodes in the texture with the same values as illustrated in Figure 3.14. Therefore, a certain level of redundancy is introduced into this data representation. These page-index textures are several orders of magnitude smaller in size than the actual data sets they represent, considering they describe the multi-resolution data representation on the level of their initial data-block subdivision (cf. Section 3.3.1).

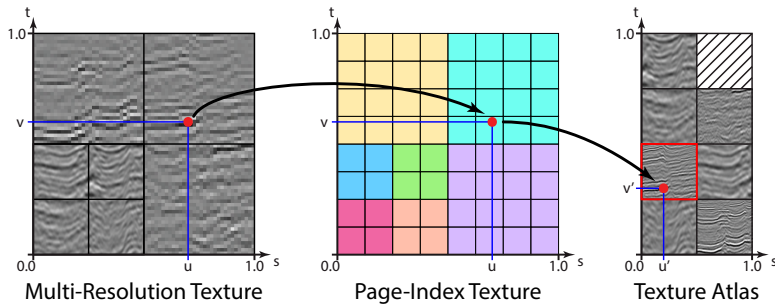


Figure 3.14: These images illustrate the procedure of a virtual texture lookup employing a page-index texture. The indirection information about the location of particular data texture pages in the atlas texture are retrieved by a lookup into the index texture using the original texture coordinates (u, v) and directly translated into the physical sampling coordinates (u', v') of the atlas texture.

A texture lookup into a virtualized texture now requires two steps. We begin by sampling the page-index texture at the requested location which produces the required indirection information and then we can use this information to transform the requested sampling position into the atlas texture coordinate system and generate the respective sample (cf. Figure 3.14). The multi-resolution data representation can be hidden completely from developers of visualization algorithms by overriding the data lookup routine of existing rendering methods with the additional dependent texture lookup.

The advantage of this approach is the retrieval of the indirection information with a constant sampling overhead at the cost of a certain memory overhead. The overhead is generated by the large areas filled with identical values representing coarse hierarchy nodes. The size of the page-index textures directly depends on the data set size and the chosen page dimensions. Smaller pages generate deeper trees and therefore require larger index textures. As the textures only store indirection information about the leaf nodes of the current hierarchy cuts, the implementation of custom filters, such as the blending of level-of-details in the hierarchy, is not directly possible. However, by adding a mipmap chain to the index texture, additional information can be stored for identifying node ancestors in the tree cuts hierarchy.

The index textures stored in GPU memory are updated when the multi-resolution hierarchy cut of the associated data set is changed. This update operation is either performed by uploading entirely new contents of index textures or by only updating texture sub-regions that are actually changed during the cut update. However, both approaches introduce comparable run-time overhead when handling a larger number of data sets. The partial updates, while allowing to transfer less data, are relying on a larger number of less efficient sub-texture updates. In order to avoid the problems of direct texture updates and to reduce the redundancy of the data transferred to the GPU, Hollemeersch et al. [HPLVdW10] demonstrated a fast index-texture update approach employing a compact change list of data pages and a geometry shader program for a direct update operation on the GPU.

While the index textures allow for straightforward access to the required indirection information for virtual coordinate translation, they present a considerable memory overhead due to the redundancy of the data representations. The working set sizes manageable in graphics memory are orders of magnitude smaller than the number of data pages of the original data sets. Consequently, when handling multiple large data sets, the degree of redundancy in single index textures is further increased as multiple data sets share the available resources.

Hierarchy Cut Serialization

The second way of representing the indirection information on the GPU is a direct serialization of the hierarchy cuts. Thereby the entire hierarchy constituting a tree cut is accessible on the GPU. As a serialization only stores information about actually existing nodes, its memory footprint is much smaller compared to a page index texture. In fact, the size of a tree-cut serialization is limited by the working set sizes, whose sizes, as discussed previously, are limited by the GPU memory actually allotted to the texture atlas. Therefore, we can rely on a fixed memory overhead for an individual serialization depending on the maximum possible cut size. The major tradeoff when using serializations of hierarchical data structures opposed to the direct access page-index textures is an increased traversal overhead for each single data lookup. Rendering algorithms, like volume ray casting algorithms, however, can amortize this cost for repeated lookups on individual pages. Such algorithms are only required to traverse the data structure when entering a new data block during ray traversal.

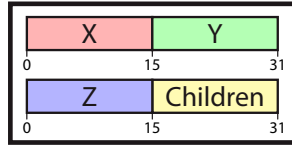


Figure 3.15: Representation of a hierarchy cut node. Two 32 bit values encode page indirection information and an offset value pointing to child nodes.

Modern graphics APIs such as **OpenGL** do not provide pointer semantics on a shader program level in order to express hierarchical data structures. For this reason, texture and buffer resources are employed to encode and efficiently traverse such data structures directly on the GPU. Although the memory overhead is manageable for hierarchy cut representations, a compact representation is still important for a GPU-based implementation. The data read during tree traversal should be minimal to increase the cache efficiency. Furthermore, smaller data structures allow for much more efficient updates in GPU memory.

The tree serialization approach in our system is based on the specific octree serialization scheme presented by Lefebvre et al. [LH05]. They proposed the use of a small 3D-texture composed of small $2 \times 2 \times 2$ volume blocks representing nodes in the octree. The individual voxels of these blocks then store specific node information and an offset value that identifies the next block in the hierarchy containing the child nodes. The result is a very narrow but very long 3D-texture encoding the octree, as the individual blocks describing nodes are strung together in one direction. Based on this data encoding, they describe an efficient traversal algorithm which walks the tree by successively transforming the virtual sampling position into the local coordinate space of the small volume blocks.

Upon experimenting with this exact approach, we very quickly encountered texture size limitations preventing us from serializing large working sets this way. In particular, the 3D-texture dimensions are limited to 2048 voxels along each axis on current GPUs. Using the described approach, we therefore can only store 1024 tree nodes under this limit. Our working set sizes, however, are potentially a multiple of this value. Consequently, we developed a quadtree and octree serialization scheme based on the proposed concept, but instead of relying on texture resources to represent the tree

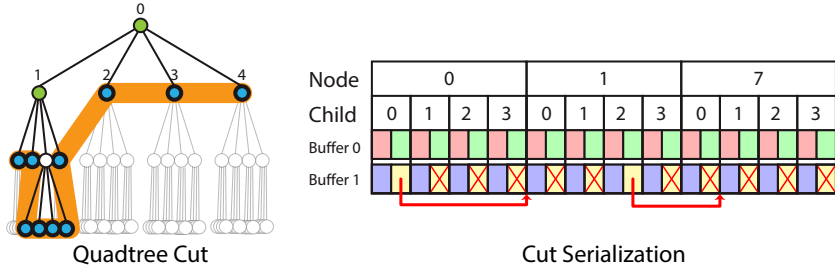


Figure 3.16: Serialization of a quadtree cut. The data structure is stored in two linear buffer resources in GPU memory. Single hierarchy-pointers address the first node of the contiguous child nodes.

cuts, we employ the ability to directly access the GPU's linear memory from shader programs. This also means that we have to adopt a different traversal approach, as our layout cannot rely on two or three-dimensional addressing schemes to select child nodes for a descent into the tree hierarchies.

We employ a compact 64 bit encoding of a single hierarchy node composed of two 32 bit words representing the atlas indirection information as well as an offset value pointing to the node's first child node (cf. Figure 3.15). A single child pointer suffices in our serialization scheme as we pack all four to eight child nodes respectively at successive positions in linear memory. It is therefore a straightforward operation to address individual child nodes. The memory layout used to store the complete serialization consists of two buffers with each buffer storing a 32 bit value of a single node. This means that a single node is represented by two values at the same offset in both buffers. Such an interleaved data organization is called *structure of arrays*, which offers improved performance as the data bandwidth of the GPU's is utilized more efficiently through coalesced memory transactions of 32 bit words [Nvi12]. We further optimize the data layout for better cache performance at run-time by considering the most common depth-first traversal order of the hierarchies. Therefore, we generate the tree cut serializations based on this order. Figure 3.16 illustrates the serialization approach for a quadtree cut.

The same serialization scheme is employed for quadtrees and octrees. The only differences in the representations are the different amounts of child nodes stored and that the quadtree nodes use the z -coordinate stored in


```
vec4
sample_texture(in sampler2D tex,
               in vec2      tex_coord)
{
    return texture(tex, tex_coord);
}
```

Listing 3.1: Plain texture lookup using intrinsic GLSL functions.

the node representation to encode the potentially used texture-array layer. Therefore, no memory overhead is created by this serialization scheme due to unused data entries.

Regarding the scalability of our chosen layouts, the use of a 16 bit-value to represent a pointer to child nodes might sound limiting. However, we store offset values to fixed size node representations in a linear memory buffer. For example a quadtree node's children take up 8 4Byte words and the offset value addresses a single of such node packs. Therefore using 16 bit representations allows us to address $2^{16} \cdot 8 = 2^{19} = 524288$ unique nodes. This value is an order of magnitude larger than currently manageable working set sizes. However, in order to scale above this amount of addressable nodes in the future, the node representation can be adapted to use less space for the actual page atlas positions to allow larger offset values. The current implementation allows us to represent the two 32 bit values constituting a node as native two-component 16 bit per component vectors without any bit-field reinterpretations at run-time.

Virtual Texture Access

In principle all data structures are accessible on the GPU to users of the virtualization system. Therefore, rendering algorithms can exploit, for instance, the octree structure of a multi-resolution volume representation to optimize the volume sampling. Other applications, however, desire the handling of the large texture resources in the same way as regular resources. Listing 3.1 illustrates a generic texture lookup using the **OpenGL** shading language GLSL. A 2D-texture is sampled using a set of texture coordinates employing an intrinsic function.

In our texture virtualization system, we provide an abstraction of the complex traversal of the serialized hierarchy cuts and the ensuing virtual

```
#include </scm/data/vtexture/vtexture.glslh>

vec4
sample_texture(in vtexture2D vtx,
               in vec2      vtx_coord)
{
    return vtexture(vtx, vtx_coord);
}
```

Listing 3.2: Virtual texture lookup using our texture virtualization embedded in GLSL.

texture coordinate translation in the form of a basic extension of the functionality of the applied shading language. This approach facilitates the easy extension of existing visualization software with the capability to handle large texture resources. Listing 3.2 illustrates a similar texture access using our abstraction layer for GLSL. By using the provided texture coordinates, representing the virtual sampling location, we traverse the internal data structures, eventually sample the physical page-atlas texture and finally return the requested data sample.

When storing not only the nodes directly associated with the hierarchy cuts but all their ancestor nodes in the associated tree hierarchy, we can also support custom trilinear and quadrilinear filtered data samples for 2D and 3D-texture data. We offer functions analog to the most frequently used intrinsic texture sampling functions, that, for example, are used to request samples at certain mipmap levels. We therefore created a drop-in replacement of the native texturing functionality allowing us to handle extremely large texture resources in a straightforward manner.

Summary

In this section we presented our approaches for the run-time abstraction of multi-resolution data representations. The working sets are stored in graphics memory employing atlas textures, assigned to the different data types handled by the data-virtualization system (e.g. height field and volume data). Our system offers two ways to represent the hierarchy cuts, constituting the working sets of individual data sets at run-time with different tradeoffs which include: a page-index texture, allowing direct access

to the location of data pages in the associated atlas texture at the cost of increased memory overhead; and a memory efficient tree serialization with an increased run-time overhead to produce the atlas indirection information. The choice of the internal tree-cut representation depends on the usage scenario. Irregular data lookup patterns benefit from the minimal run-time overhead of the index texture, while advanced rendering algorithms, such as volume ray casting, can exploit the tree serialization and amortize the additional traversal costs over larger amounts of data lookups. Independent of the chosen tree-cut representation, our system offers completely virtualized access to the multi-resolution data sets, allowing users to work with extremely large data sets in the same way as regular texture resources.

3.5 Working Set Generation

The working sets of data pages stored in GPU memory represent the only sub-set of data available for rendering. They, therefore, need to be carefully selected to visually represent the original data model. In order to find an appropriate multi-resolution hierarchy cut, so-called *selection algorithms* are employed. They find the best approximation of the data set towards a defined heuristic selection criterion. Such heuristics are defined on the basis of, for instance, specific points of interest, the screen-projection size or viewer distance of a data sub-block or data-based criteria [CF11]. The implementation of selection algorithms primarily considers memory size budgets the generated hierarchy cuts are required to conform to, but they also need to consider additional system parameters such as a data bandwidth budget for updating the working sets on the GPU. Such bandwidth budgets are a way of preventing the temporary stalling of real-time rendering methods in case of large working set changes resulting from extensive cut adjustments.

The actual selection criteria are generally expressed through a monotonic priority assigned to each node in the quadtree and octree hierarchy. A priority function is called monotonic if the priority assigned to child nodes is less than or equal to their parents' priority. Typical basic selection algorithms employ a priority queue to store nodes that are candidates for refinement [BNS01, GWGS02]. Generally this queue is initialized with the root node of a hierarchy. The hierarchy cut is then constructed by removing the node with the highest priority, splitting it and inserting all its non-empty

children back into the queue. This is repeated as long as there is still texture memory available for refinement and a designated page upload budget is not exceeded. The remaining nodes in the priority queue represent the current tree cut and therefore define the working set of pages to store in graphics memory. In order to support custom filtering algorithms of multiple levels of detail during rendering, this working set can be expanded to include all data pages associated with the hierarchy constituting the tree cut. In such cases the selection algorithm needs to consider the additional storage of the inner hierarchy nodes with relation to the memory budget. The additional storage of these nodes effectively reduces the space for higher levels of detail by at most 33% for quadtree cuts and 14% for quadtrees.

The basic approaches rebuild the hierarchy cuts from the ground up with every invocation of the selection algorithm. They ignore frame-to-frame coherence information such as the currently selected cut. It is desirable to consider the information represented by the current cut under the assumption that the assigned node priorities only gradually change, when for instance the viewing position is moved. Through an incremental refinement of the current tree-cut, memory bandwidth limits can be considered. By only allowing a certain amount of new nodes to be inserted into the cut, extensive transfer times of data from the host to the graphics memory can be avoided, hence preventing the stalling of the rendering pipeline and therefore undesired variations in the rendering frame rate.

More advanced selection methods, such as the one presented by Duchaineau et al. [DWS⁺97], employ a dual-queue strategy extending the basic split-only approaches to split/merge algorithms. This allows for fine-grained incremental updates to be made to an existing hierarchy constituting a tree cut. The first priority queue still stores all nodes of the current cut that can be further refined - the *split candidates*. The second queue stores nodes that can be directly merged, which are parent nodes of exclusively leaf nodes of the tree cut hierarchy. These nodes are the *merge candidates*. The merge queue is sorted in descending order according to the node priorities. The algorithm starts by initializing the split-candidates queue with the root node of the particular tree. It then performs a series of split operations until either the memory or bandwidth budget for this algorithm invocation is reached. When the memory budget is reached but nodes still can be transferred, the algorithm checks the merge queue. If the priority of the first merge candidate is less than the priority of the current split candidate then the node is merged, thus freeing up memory resources for new split

operations to refine the cut. Such greedy-style algorithms approximate a cut selection based on locally available information [CF11]. By exploiting frame-to-frame coherence a limited amount of split and merge operations are performed to successively reach a cut selection over a number of rendering frames.

In the following sub-sections, we detail our approach of a unified selection algorithm for multiple tree cuts constituting quadtree and octree multi-resolution data sets. The algorithm follows a dual-queue split/merge approach relying on a monotonous priority function global to all data sets. We conclude this section with design choices for the priority function facilitating implicit data pre-fetching and discuss different approaches to the generation of the actual node priorities.

3.5.1 Unified Selection Algorithm

We adopted a dual-queue split/merge algorithm in our data-virtualization system for the concurrent update of multiple multi-resolution hierarchies. The challenge is to update multiple hierarchy cuts considering fully global system limits as well as more local constraints. System global limits are represented by the restricted bandwidth for the update of the working sets on the GPU. The local constraints refer to the memory constraints of virtual texture contexts shared by data sets of a common type.

Running the selection algorithm sequentially for each single data set in our system is not a viable option. Such an approach would lead to a first-come, first-served problem of a small number of data sets using up most of the memory and bandwidth budgets. Another solution would be to assign individual split and merge queues to all data sets but run the selection algorithm only one step at a time and decide which data set is allowed to execute the next operation. This requires a global prioritization metric to base such scheduling decisions on common criteria.

Our solution reduces the scheduling overhead of the selection algorithm by also utilizing a global prioritization metric and, more importantly, single split and merge queues for storing different node types. In fact, at the level of the selection algorithm we only need to deal with two types of nodes: quadtree and octree nodes. The only operations the algorithm is required to perform on instances of nodes are splitting (replacing the node by its children) and merging operations (replacing a set of leaf nodes by their

parent node), thereby facilitating an abstract definition of the selection algorithm agnostic of the actual node type.

The algorithm is typically invoked once during each rendering frame of a visualization system based on our data-virtualization system. The following overview illustrates the basic steps the algorithm performs during each invocation:

1. Initialize split and merge queues:
 - a) Update the priorities of all nodes in all current hierarchy cuts.
 - b) Insert splittable nodes of all current cuts into split queue.
 - c) Insert mergeable nodes of all hierarchies constituting tree cuts into merge queue.
2. While global bandwidth budget not reached and splits still viable:
 - a) If memory budget of current split nodes texture context reached:
 - i) Perform merge operation in context.
 - b) Split current split candidate node.

The above abstract description does not consider that nodes selected for upload to the GPU might not be available in the page cache in host memory. When considering a split candidate node, we therefore first check if all its children are actually present in the respective page cache. Upon detecting missing children, the associated data pages are requested to be asynchronously loaded from external storage and the split operation is suspended and the algorithm moves to the next split candidate. Consequently, the required split operation is deferred until the required data is fetched from external storage. This way neither the update mechanism nor the rendering is stalled because of missing data. Furthermore, all data pages representing inner nodes of the current hierarchies which in turn represent the tree cuts in the working set are stored on the GPU. This has two advantages. Firstly, it allows advanced filtering algorithms of multiple levels of detail during rendering, and secondly, it allows instantaneous merge operations during hierarchy updates. Without this approach, merge operations would also need to be deferred when the data page of the respective parent node is not available in the page cache.

The run-time complexity of the initialization step of the algorithm is dependent on the maximum tree-cut sizes M , defined by the memory-budget size available to store the working sets of all handled data sets.

This number is orders of magnitude smaller than the actual numbers of data pages constituting the handled data sets N . The initial insertion of all prioritized nodes into the priority queues can be performed using $\mathcal{O}(M)$ time complexity, based on the typical binary heap implementations of priority queues. The actual update phase is limited by the maximum amount of data pages B , which are allowed to be uploaded to the GPU in one rendering frame. This number is again orders of magnitude smaller than the maximum cut sizes M . The number of queue node insertions into the priority queues during the update phase is at most B data pages. With a single insert operation into a priority queue typically performed with $\mathcal{O}(\log M)$, the update phase exhibits a time complexity of $\mathcal{O}(B \log M)$. As a result, the entire algorithm exhibits a run-time complexity of $\mathcal{O}(M + B \log M)$. As $B \ll M \ll N$ represents the relation of the important run-time parameters, the complexity of our algorithm is dominated by the initialization phase of the two employed priority queues.

3.5.2 Implicit Page Pre-Fetching

With growing memory sizes on modern GPUs, the space available to store working sets of individual data sets increases accordingly. As a result, more data pages can be stored in the respective atlas textures than are actually required for an adequate visual representation. This, of course, heavily depends on the data set type. While volumetric data sets usually take up all memory made available to them, two-dimensional data sets often leave free memory after selecting a suitable working set according to the employed selection criterion. This additionally available memory can be used to pre-fetch data pages to support roaming through large data sets. A possible use case are zoom-in operations, where additional data to the actually required hierarchy cut can be used to smoothly blend into the finer levels of detail as they become visible. Therefore, so-called level-of-detail pop-in artifacts can be reduced through the use of data pre-fetching.

We designed the selection algorithm in our system to implicitly pre-fetch data pages whenever, according the used selection criterion, suitable multi-resolution hierarchy cuts are achieved and memory resources are still available on the GPU. This means that only after all actually required data pages are part of the particular tree-cut hierarchies, these cut hierarchies are extended with data pages that might get used in future rendering frames. We achieve this behavior through additional requirements to the employed

monotonous priority function. The selection metric needs to ensure that an actually required data page is assigned a priority that is at all times larger than a priority of a data page designated for pre-fetching. This property ensures that all available memory resources on the GPU are utilized through the unified selection algorithm, while maintaining the monotony property of the prioritization metric. Furthermore, it ensures that the implicit pre-fetching is only engaged when all required data is already part of the working sets represented by the tree-cut hierarchies.

3.5.3 Page-Priority Generation Approaches

The split/merge algorithm is driven by a global, monotonous priority function. This function represents the main facility necessary to steer the hierarchy-cut selection process in our approach. It therefore needs to encode all criteria regarding data visibility, required data resolution or special points of interest in the data sets. Two basic approaches exist for generating view-dependent information concerning the required levels of detail of a multi-resolution representation: heuristic methods and direct feedback mechanisms. Heuristic methods try to determine the required levels of detail through view-dependent or data-dependent criteria. Feedback mechanisms collect information about the required data directly during rendering, which is then used by the selection algorithm. In our system, the priority information is either generated through a view-dependent heuristic or by directly employing a feedback mechanism during the rendering process on the GPU.

Through the evolution of our system, during the research work constituting this thesis, we started with a basic, heuristic view-dependent object-space metric relying on node distances to the viewer similar to LaMar et al. [LHJ99]. This approach requires book-keeping for tracking the positions of all data blocks representing data pages in three-dimensional space. While this tracking works in a straightforward way when displaying data sets using their regular geometry, such as a direct volume rendering, it is more difficult to track the actual positions of two-dimensional data blocks of height-field data sets or volume data mapped to arbitrary geometric primitives (cf. Section 3.1.1). These application scenarios, however, are quite common in geo-scientific visualizations as single large volumetric or image primitives often get mapped to arbitrary height-field horizon surface geometries, largely preventing the analytical tracking of required data pages.

Furthermore, such basic object-space heuristic approaches do not directly consider data visibility. Additional occlusion culling methods are required to generate the required visibility information. Therefore, not only is a certain amount of run-time overhead added to the system, but rendering algorithms based on a multi-resolution data-management system must be specifically adapted to support visibility detection algorithms. Proposed rendering algorithms especially tailored to exploit hardware-accelerated occlusion tests illustrate the complexity of the inclusion of such approaches for the visualization of bricked volumes [GMG08] and tiled horizon geometries [PGSF04]. With our system design, we want to provide a data-virtualization system that offers minimal impact on existing frameworks and minimal impact on the definition of new rendering algorithms based on extremely large data sets.

For our out-of-core data-virtualization system, we employ a direct feedback mechanism which determines required data pages directly during rendering, derived from the actual data usage. By either exploiting inherent hardware features, such as the ubiquitously available z-buffer, or by using custom image-order rendering approaches, such as ray casting, our system is capable of resolving data visibility without the application of any additional occlusion culling approaches.

3.6 Level-of-Detail Feedback

The main goal of a direct feedback mechanism is to collect the inherently available information about required data pages directly during the rendering process. Most information about data usage and visibility is available at the fragment processing level in the rendering pipeline. Through hardware features of modern rasterization pipelines which are implemented in current generations of GPUs (e. g. discarding of geometric primitives in the geometry stages or early-z fragment culling), much information is readily available that could inform the tree-cut selection method regarding which data blocks actually made it to the fragment processor and are therefore potentially visible in the final frame buffer. When applying image-order rendering approaches, such as the ray casting-based methods proposed in Chapter 4, data access is potentially restricted to actually visible data.

A basic approach for the generation of feedback information from the rendering pipeline is to store additional data-usage information to the

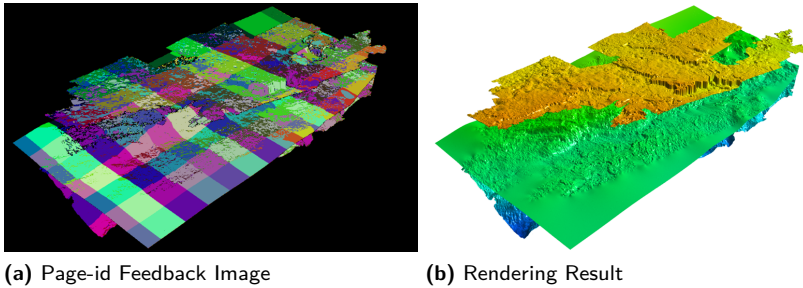


Figure 3.17: This figure shows the results of a level-of-detail feedback mechanism, depicting a height-field tile index per pixel, during the rendering of three layered horizon surfaces. (a) illustrates the data-page identifiers of the required height-field tiles, determined by a level-of-detail estimation heuristic, color-coded on a per-pixel basis. (b) shows the associated rendering of three stacked seismic horizons using three virtualized height-fields.

frame buffer. Goss et al. [GY98] proposed an extension to the standard graphics pipeline, expanding the frame buffer with a per-pixel texture-tile index in addition to the existing color and depth buffer. This index is used in their approach to store usage information about actually visible parts of a tiled texture-mipmap pyramid. While the proposed extension is not part of current hardware designs, its behavior can be emulated through multiple render-target setups on modern GPUs. Such an approach facilitates the generation of per-pixel information about visible data sub-blocks by exploiting the hardware z-buffer to write out data-usage information of visible pixels. The information about the actually required levels of detail contained in these feedback images is then analyzed and evaluated to steer the multi-resolution cut selection method. However, this requires the transfer of the information to the CPU performing the actual hierarchy-cut updates. As shown in Figure 3.17, a resulting feedback render-target image potentially contains a lot of redundant information, as data pages cover larger screen areas. In order to minimize feedback-data read-back latency and processing times, direct feedback methods typically employ separate pre-rendering passes to generate the feedback usage information in smaller off-screen buffers [Bar08, HPLVdW10]. While such pre-rendering passes are

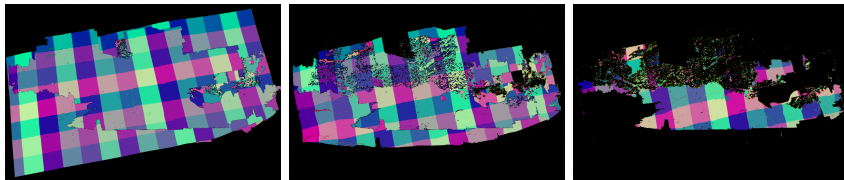


Figure 3.18: This figure illustrates the use of three fixed feedback image layers for the ray casting of three translucent horizon surfaces. The images represent level-of-detail feedback requests for the first three surface intersections of the viewing rays.

typically performed without the application of costly illumination models, they still present an additional run-time overhead, especially considering costly direct volume rendering applications.

Feedback approaches employing a single feedback image buffer are only able to generate per-pixel information about a single required data page. The translucent display of data sets, however, requires multiple per-pixel feedback slots and is therefore not supported by these methods. The prime example for visualization methods heavily depending on a variable number of feedback slots is direct volume rendering. Crassin et al. [CNLE09] extended the feedback-driven approaches to translucent volumes by employing multiple auxiliary render targets to store spatially and temporally subsampled level-of-detail usage information. This approach is limited to a fixed number of feedback slots, which are amortized over larger screen-tiles and multiple rendering frames. Still, this approach exhibits an upper limit for the scalability of larger data sets. The fixed allocation of feedback slots to certain viewport pixels prevents the reassignment of memory resources from screen portions with lower demand to portions with higher demands. Figure 3.18 illustrates the memory over and under-allocation for the ray casting-based translucent visualization of the three horizon surfaces depicted in Figure 3.17b. Only the first three surface intersections of a viewing ray, associated with a viewport pixel, and the translucent surfaces are able to generate a feedback request of an appropriate height-field tile. Due to the fixed per-pixel feedback-slot assignment of such an approach, this leads to a memory under-allocation where additional intersections occur along the ray and over-allocation in areas with few or no intersections. A variable alloca-

tion of feedback resources is key for efficiently communicating information about required data blocks for more complex models containing multiple translucent horizon surfaces and volume primitives.

3.6.1 Process Overview

The feedback mechanism proposed in this work facilitates a variable feedback resource allocation on the basis of per-pixel linked lists, generated on the GPU directly during the rendering process. This feedback mechanism is subdivided into two principal stages:

1. **Feedback data generation:** This stage is solely concerned with collecting information about required data pages during the rendering process. The following two steps are performed upon accessing a virtualized data set at a particular sampling location:
 - ▶ *Level-of-detail estimation:* The data block of the associated multi-resolution hierarchy is determined, which would best represent the data set under the current viewing and rendering properties (e.g. viewer position, zoom level, output resolution). In our system, this level-of-detail estimation is accomplished using a texture space metric depending on the provided set of virtual texture coordinates.
 - ▶ *Feedback list generation:* The resulting information about the required data block for the particular data lookup is encoded and appended to a linked list associated with the location of the particular output pixel. Because of the highly parallel nature of current GPUs, a special algorithm is employed for the concurrent generation of the per-pixel linked lists.
2. **Feedback evaluation:** After collecting all feedback information during a rendering frame, this stage prepares the feedback information for use in the multi-resolution hierarchy update mechanism running on the CPU. However, due to the large amount of feedback information potentially generated during the rendering of geological data sets (especially when visualizing translucent seismic volumes), it is unfeasible to transfer and evaluate the entire data volume on the CPU. Therefore, the evaluation stage is split into two sub-stages:
 - ▶ *Feedback compression:* In order to reduce the amount of information required to be transferred from GPU memory to the CPU after each

rendering frame, we employ a compression method running directly on the GPU. This method generates a compressed list of required data pages associated with the quantity of their occurrence in all feedback lists of all viewport pixels.

- *Feedback evaluation:* The compressed list of required data pages is finally read back to the CPU where the pixel quantity information associated with each entry is used in the prioritization function driving the update of the multi-resolution representations of the virtualized data sets.

In the following parts of this section, we present our design decisions and implementation details of the individual stages of our level-of-detail feedback mechanism. First, we describe the internal data encoding of the feedback data, before detailing the construction and representation of the linked list structure on the GPU. We then present the feedback compression approach to reduce the amount of data required to be transferred back to and evaluated on the CPU. We finally describe the construction of a monotonous priority function based on the acquired level-of-detail feedback information, steering the actual multi-resolution hierarchy-cut selection algorithm.

3.6.2 Feedback Data

Before detailing our choices for the direct feedback mechanism, we present the actual feedback data which encodes the information about required data pages. A data page in our system, representing data sub-blocks of two-dimensional height fields or three-dimensional volume data sets, is identified by its page id consisting of an instance id of the associated data set and a linear index derived from the z-order curve, as described in Section 3.3.3. This page id is encoded as a 32 bit unsigned integer value, used throughout the memory management system executed on the host system. During run-time on the GPU, upon determining a required data sub-block, we do not have direct access to page ids of the associated data pages and need to generate them on demand.

Level-of-Detail Estimation

The estimation of the actually required data sub-blocks and therefore data pages in our system is based on a texture space metric derived from the calculation of the level of detail in a mipmap image hierarchy [SA12]. Given a set of non-normalized texture coordinates $\mathbf{c} = (u, v, w)$ and a rasterization fragment coordinate (x, y) , the required level of detail $\lambda(x, y)$ is calculated by:

$$\lambda(x, y) = \log_2(\rho(x, y)) \quad (3.7)$$

$$\rho(x, y) = max \left\{ \sqrt{\left(\frac{\delta u}{\delta x}\right)^2 + \left(\frac{\delta v}{\delta x}\right)^2 + \left(\frac{\delta w}{\delta x}\right)^2}, \sqrt{\left(\frac{\delta u}{\delta y}\right)^2 + \left(\frac{\delta v}{\delta y}\right)^2 + \left(\frac{\delta w}{\delta y}\right)^2} \right\} \quad (3.8)$$

This level-of-detail estimation is calculated on the GPU by using intrinsic functions providing the partial derivatives of the texture coordinates in screen-space based on the pixel position. The position \mathbf{p}_{pos} and level \mathbf{p}_{level} of the required data block in the associated multi-resolution hierarchy \mathbf{t} is calculated based on the returned level of detail $\lambda(x, y)$ and the maximum height of the multi-resolution hierarchy $\mathbf{h}(\mathbf{t})$:

$$\mathbf{p}_{level} = \lceil \mathbf{h}(\mathbf{t}) - \lambda(x, y) \rceil \quad (3.9)$$

$$\mathbf{p}_{pos} = \lfloor \mathbf{c}_{norm} \cdot 2^{\mathbf{p}_{level}} \rfloor \quad (3.10)$$

While the texture coordinate \mathbf{c} is required in the non-normalized range from 0.0 to the actual data set resolution for the calculation of $\lambda(x, y)$, it is normalized for the calculation of the position \mathbf{p}_{pos} of the required data block in the multi-resolution hierarchy.

Unified Page Identifier

The information resulting from the level-of-detail estimation in combination with the associated data-set instance id can be used to communicate the required data pages back to the host system for the multi-resolution hierarchy update mechanism. In order to allow the direct usage of the information transferred to the host system, we employ the same page-id encoding used on the CPU (cf. Section 3.3.3). This approach presents the major advantage that the page ids are a unified representation of data pages associated with quadtree and octree nodes throughout the entire data-virtualization system

on the CPU and the GPU without an additional translation of the feedback information on the CPU for the feedback evaluation.

Therefore, we employ the same z-order curve encoding of the page index directly on the GPU. The hierarchy position and level of a data page are encoded into the z-order curve index employing fast bit-manipulation operations directly on the GPU, resulting in a 32 bit unsigned integer value. This value is then combined with the instance id of the associated data set to form the final page id. As a result, we obtain a very compact feedback encoding, which is independent of the associated data set type, with unique page identifiers on the GPU and the CPU side of the memory management and texture virtualization system.

3.6.3 Concurrent Generation of Linked Feedback Lists

With the recent development of modern GPUs, today we are able to much more freely read and write GPU memory. Features, such as scattered writes to texture and linear memory and atomic arithmetic on data values shared between multiple parallel processing units on the GPU, facilitate the implementation of advanced parallel data structures on the GPU. Yang et al. [YHGT10] demonstrated an implementation of per-pixel linked lists for the use in order independent transparency rendering of polygonal models.

Our direct level-of-detail feedback mechanism employs an implementation of per-pixel singly linked lists to collect a variable amount of page ids about required data pages during rendering. In order to save memory resources, these lists are not strictly generated for each pixel of the viewport, but we allow a configurable regular sub-sampling of the viewport. This enables us to save precious GPU memory and computing resources for the list storage and generation.

Concurrent List Generation

The basic representation of concurrent per-pixel singly linked lists on the GPU is defined by two memory resources: a head pointer texture image and a linear list-node buffer. The head pointer image stores the beginnings of the lists for the output pixels. Its size is identical to the viewport extents or the desired fraction of the viewport size for sub-sampling purposes. The list-node buffer then contains the data associated with each list node (the payload) as well as a link to the next list node. For the creation of new list

nodes, an additional global counter is employed during list-insert operations, pointing to the next free node in the list-node buffer. All these resources are used globally by all fragment processing units on the GPU. Due to the highly parallel nature of modern GPUs, access to these shared global resources is highly concurrent. Therefore, access, especially to the global node counter, is handled by atomic operations, securing consistent access to the global list-node buffer upon concurrent list-insertion operations. For the initialization of the data structures, the counter is cleared to 0 and the head pointer image is filled with a constant value representing the end of the list (EOL). The list-node buffer is not required to be assigned special values.

The lists are typically generated by inserting new elements at the front. This has the advantage that the insert operations are performed in $\mathcal{O}(1)$ time as opposed to linear time complexity for back-insertions. The insertion of an element into a list associated with an individual pixel is accomplished by the following steps:

1. Generate a new node pointer by atomically reading and simultaneously incrementing the global node counter.
2. Atomically exchange the new node pointer with the head pointer stored in the head image.
3. Store the node data at the new node location and insert the old head pointer as the link to the next node.

This basic approach has the disadvantage in that it requires two atomic operations per insert operation. Furthermore, it exhibits a certain memory overhead for the storage of the node links. The storage of page id data of requested data blocks in the list nodes requires two values: one storing the unique page id and a second one storing the link pointer to the next list node. Atomic operations present crucial synchronization overhead. Different fragment processing units accessing a single globally shared value, such as the node counter or a single head image entry, are scheduled to wait upon the completion of the access of other processing units.

Paged Lists

A solution to reduce the reliance on atomic operations and simultaneously reduce the memory overhead of the node link storage is to transform the

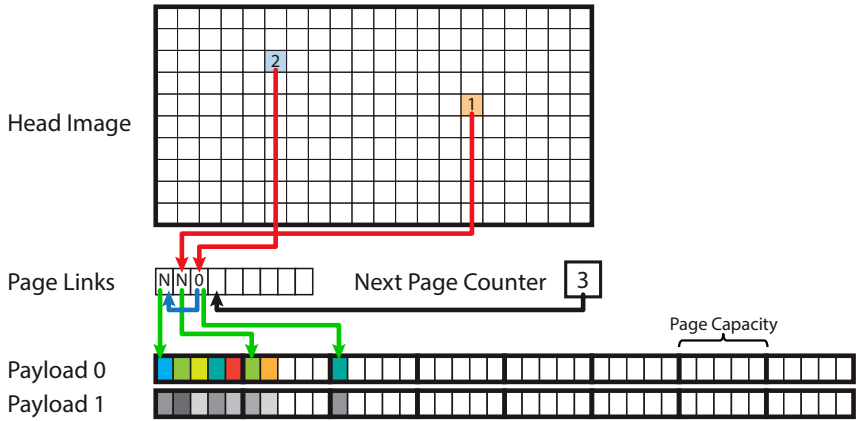


Figure 3.19: This image illustrates the representation of per-pixel paged singly linked lists. The head image holds the pointers to the beginning of the lists. The list-page links buffer stores the link pointers to connected memory pages. The payload buffers contain the actual feedback data stored in the lists.

current list of single entries per list node into a list of pages containing multiple entries per node. A similar approach was presented by Crassin [Cra10] as an extension to the rendering approach for order independent transparency of Yang et al. [YHGT10].

The approach subdivides the list payload buffer into fixed size list pages, which now exclusively store the associated data. An additional buffer then stores the link pointers and information about the available capacities of the particular list pages. A new list page is only inserted into a list when the currently used list page overflows. This way the overhead of additional atomic operations for the list-page insertion operation is amortized for multiple data insertion operations. However, as still multiple fragment processors are able to work on the same fragment (e.g. overlapping geometric primitives concurrently processed on the GPU), access to the individual list of a single pixel must be secured against concurrent access. This is achieved through emulating a critical section guarded by a mutual exclusion (mutex) by means of atomically setting a flag in the head image. The operations executed inside the critical section solely encompass access to the head

image and retrieval of a new write position in the list payload buffer. Only if a new list page is required, additional atomic operations on the list-node counter and head image are also required. Therefore, paged concurrent lists offer reduced synchronization overhead related to atomic operations on shared resources as well as reduced memory overhead for the storage of the list-link pointers. Figure 3.19 illustrates the data structures involved in the representation of concurrent paged singly linked per-pixel lists on the GPU.

While a paged list implementation offers advantages during run-time and intensive utilization, it introduces memory overhead when only lightly used. Upon inserting a single element into a list, a full list page is allocated for the associated pixel. The capacity of the list pages therefore needs to be carefully chosen to match the application scenario.

Extended Page Information

Our paged list approach further offers the ability to store additional data associated with each data-page id inserted into the lists, through supplemental payload buffers as illustrated in Figure 3.19. This additional information can be utilized during the evaluation and hierarchy-cut selection to weigh different required data pages on the basis of, for instance, the currently accumulated opacity along viewing rays or depth information. A potential application scenario is direct volume rendering wherein volume bricks might be just faintly visible through only lightly translucent volume regions displayed in front of them. Consequently, additionally stored information about residual transparency can be utilized to steer the octree-cut selection to different, much more apparently visible volume sub-blocks.

Explicit Feedback Requests

The actual generation and list insertion of feedback requests is accomplished in one of two ways in our data-virtualization system. Either the feedback requests are appended implicitly upon each virtual texture access, requiring no further action by a developer (cf. Listing 3.2 on page 78), or the requests are generated and appended explicitly by the developer as shown in Listing 3.3.

This approach offers reduced feedback generation and storage overhead for use cases where multiple samples are taken around a central sampling location, such as the calculation of a local data gradient. The required level

```
#include </scm/data/vtexture/vtexture.glslh>

vec4
sample_texture(in vtexture2D vtex,
               in vec2   vtex_coord)
{
    vtexture_append_request(vtex, vtex_coord);
    return vtexture_sample(vtex, vtex_coord);
}
```

Listing 3.3: Virtual texture lookup and explicit level-of-detail request using our texture virtualization embedded in GLSL.

of detail is then solely determined and requested for a single data sample instead of a larger amount of samples. Another example of techniques based on our texture-virtualization system which greatly benefit from explicit feedback generation are volume ray casting algorithms. By issuing requests on the volume-brick level instead of for each sample taken during the ray traversal, the feedback is only generated for larger volume sub-blocks (cf. Figure 3.20). Thereby the redundancy in the feedback lists and the required list-payload storage is minimized.

Adaptive Memory Allocation

The memory resources implementing the list-link and payload storage of the feedback lists are pre-allocated in GPU memory from the host system before actual usage, as current GPUs do not offer dynamic memory management from the device side. Consequently, on system start-up the memory resources are initialized to typically fit a single list page per feedback pixel. In order to adapt to an increase in required list capacity, the atomic list-page counter is read back asynchronously to the CPU after each rendering frame. If this counter overflows the allocated amount of list pages and therefore prevents additional list-insertion operations, the buffer resources are enlarged accordingly. This approach results in few rendering frames of partial feedback before the asynchronously received information is attended to and the increased payload capacity is available for use. However, when employing rendering approaches based on a front-to-back ordering of the data accesses (e.g. volume ray casting), such partial feedback is not critical in the feedback evaluation process as the most prominently visible parts of

the scene are able to generate and store feedback requests. In the reverse case, upon detecting a consistent under-use of the available resources, the buffer storage is reduced in order to minimize the memory overhead on the GPU. The size of the memory resources representing the list storage on the GPU also directly affects the run-time overhead of the feedback evaluation and compression stage performed on the GPU, which is described in the following parts.

3.6.4 Feedback Evaluation

The fundamental purpose of the direct level-of-detail feedback mechanism is to provide information to the multi-resolution hierarchy-cut selection algorithm about required, potentially visible data pages. Furthermore, this information needs to provide a way of prioritizing the required data pages to steer the selection algorithms' decisions on what pages to insert into the multi-resolution hierarchy cuts and therefore upload to the GPU (cf. Section 3.5.1).

Existing approaches relying on feedback information collected during rendering by Hollemeersch et al. [HPLVdW10] and Crassin et al. [CNLE09] simply determine if a data page is visible, and they do not generate any prioritization among different pages. With our system we follow the basic idea presented by Goss et al. [GY98] to count the occurrences of individual requested data pages in the generated feedback lists to steer the hierarchy updates. Data sub-blocks of virtualized height fields or volumes covering larger areas in the viewport are classified with a higher priority than blocks only displayed on smaller areas. As a result, when detecting missing data pages in the current working set on the GPU, the associated data blocks with the potentially biggest impact on the final rendered image are selected earlier for loading to the GPU. The feedback evaluation process therefore fundamentally needs to traverse the generated per-pixel lists and count the occurrences of individual requested data pages.

Evaluation Approaches

The evaluation of the feedback information poses a challenge related to the amount of information generated and required to be processed. For example, the visualization of a virtualized seismic volume shown in Figure 3.20 uses a viewport resolution of 1024×768 with a 2×2 sub-sampling of the viewport

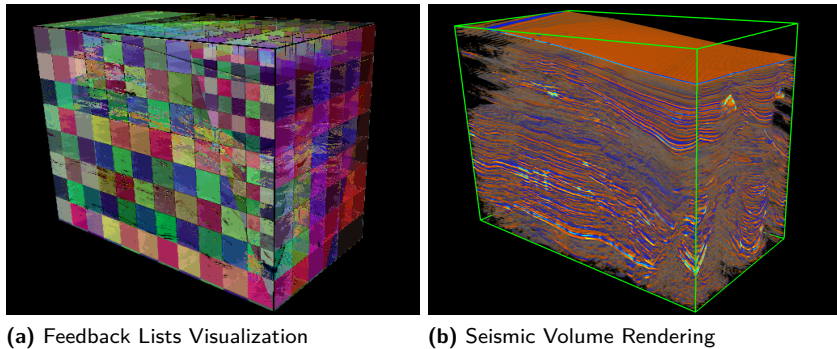


Figure 3.20: Visualization of per-pixel feedback lists generated during a direct volume rendering of a seismic volume. (a) illustrates the generated feedback information through pseudo-colors representing the requested volume data blocks. The feedback is generated and collected through a volume ray casting-based rendering of a seismic volume shown in (b).

for the feedback generation, which results in 512×384 possible feedback lists. The capacity of the payload buffer for the depicted viewing position and employed transfer function was 690 thousand list pages with a capacity of four entries. This resulted in a payload-buffer size of 10.6 MiB. Different usage scenarios employing larger screen resolutions and transfer functions require even larger resources on the GPU (cf. practical results presented in Section 3.8.2). A naïve approach would read back the data constituting the fragment lists to the host system after each complete rendering frame and evaluate it on the CPU. However, the data transfer of the information and its evaluation poses a large run-time overhead.

While the transfer can be handled efficiently through asynchronous transfers from the GPU, in our experience the CPU-bound evaluation poses the biggest overhead preventing real-time operation for large data sets. A compression of the GPU-bound information can greatly benefit the data transfer and evaluation process. Hollemeersch et al. [HPLVdW10] presented an approach for the compression of a single-layered feedback image using an evaluation stage executed directly on the GPU. The presented approach generates a compressed list of required data pages for a single virtualized 2D-texture data set. They, however, discard any information about the

actual quantity of data-page requests. Their approach works by utilizing a large page table stored in GPU memory constituting the multi-resolution quadtree hierarchy of the handled data set. This page table contains an entry for each quadtree node. After each rendering frame, the information stored in the feedback image is evaluated and encountered data pages are marked as required in the page table. After finishing this first evaluation step, the table contains only a few active entries for the actually occurring data pages. This information is then condensed into a compact list of data pages requested during the rendering process. The list is finally read back to the CPU to update a multi-resolution representation of the virtualized data set.

We evaluated a similar approach for the feedback compression in our system, but quickly discovered that it represents no viable solution for multiple, extremely large data sets. Firstly, consider the size of the required page tables on the GPU: they directly depend on the sizes of the quadtrees and octrees representing the large height fields or volumes in our system. For example an octree with 8 levels is composed of roughly 19.2 million nodes representing potential data pages. The required page table stored in GPU memory for the described evaluation process therefore requires 73.1 MiB to store individual entries for each octree node. Secondly, in order to not only set a simple request-flag but to count the actual occurrences of a single data page, the data access of the employed kernels must be implemented as atomic increment operations. Furthermore, these operations access the memory storing the page table extremely sparsely and exhibit frequent access collisions at a single memory location, caused by the parallel processing of redundant feedback information. The performance we observed, depending on the number of simultaneously handled data sets and their extents, does not permit a real-time application of this approach to multiple large data sets.

Histogram Compression

Enabled by the use of unified page identifiers to represent data sub-blocks of different handled data sets on the GPU, a different compression approach is viable, forgoing special page resources dedicated to individual data sets on the GPU. The basic idea of our approach is to generate a histogram over all page identifiers stored in the feedback lists. This histogram then eventually contains only entries for actually occurring data pages in the feedback lists

and therefore represents the desired compact representation of a list of requested data pages associated with the quantity of their occurrences.

The actual feedback information in our approach is stored in the payload buffer independent of any structural information concerning the composition of the individual lists. As a result, the access to the feedback information is greatly simplified. We base the histogram generation process on the evaluation of the linear memory sequence represented by the payload buffer as opposed to the independent lists. The GPU-based creation of the histograms therefore consists of the following two steps:

- Sort the input sequence of page ids stored in the list payload buffer.
- Compact the sorted sequence of page ids by eliminating and counting duplicate subsequent values.

The creation of a histogram directly on the GPU is typically implemented through parallel prefix-sum scan algorithms [HSO07]. The efficient GPU-based implementation of such algorithms, however, would exceed the scope of the thesis at hand. We therefore based our histogram compaction approach on building blocks offered by the Thrust parallel algorithms library [BH11]. This library offers a standard interface for parallel algorithms implemented on top of the CUDA GPU-compute API for NVIDIA GPUs. Consequently, the feedback mechanism in our system follows a staged execution on the GPU, where the feedback data is generated and eventually compressed before transporting a compact representation to the CPU for the final feedback evaluation and working set updates as illustrated in Figure 3.21.

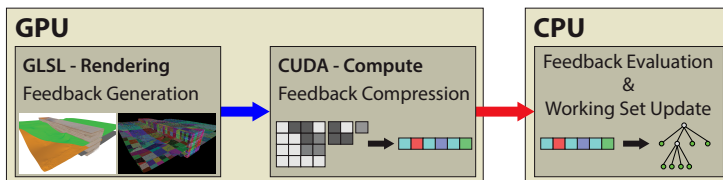


Figure 3.21: This figure illustrates the execution of the different stages of the level-of-detail feedback mechanism. The first stages are executed directly on the GPU using GLSL for the level-of-detail request generation during the rendering process and CUDA for the feedback compression before transferring the data to the CPU for the feedback evaluation and working set update process.

```

void
thrust_compact(uint32*   tmp_pids,           // temp buffer
               uint32*   tmp_pcov,          // temp buffer
               uint32*   in_pids,           // input page ids
               int32      in_pids_size,     // input size
               uint32*   out_pids,          // output page ids
               uint32*   out_pcov,          // output page coverages
               int32*     out_hist_size)    // output size
{
    using thrust::device_ptr;
    using thrust::device_pointer_cast;

    device_ptr<uint32> hist_pids = device_pointer_cast(tmp_pids);
    device_ptr<uint32> hist_pcov = device_pointer_cast(tmp_pcov);
    device_ptr<uint32> d_pids     = device_pointer_cast(in_pids);
    std::size_t        n_pids     = 0;

    // sort page ids
    thrust::sort(d_pids, d_pids + in_pids_size);

    // generate histogram of sorted page ids
    n_pids = thrust::reduce_by_key(
        d_pids, d_pids + in_pids_size, // input key sequence
        thrust::constant_iterator<uint32>(1u),
        hist_pids,                       // output key sequence
        hist_pcov,                       // output value sequence
        ).first - d_pids_comp;           // compute the output size

    // clear page id buffer
    thrust::fill(d_pids, d_pids + in_array_size, 0xffffffffu);

    // copy resulting histogram buffers to host
    *out_hist_size = static_cast<int32>(n_pids);
    thrust::copy(hist_pids, hist_pids + n_pids, out_pids);
    thrust::copy(hist_pids, hist_pids + n_pids, out_coverages);
}

```

Listing 3.4: Generation of page-id histograms using the Thrust parallel algorithms library.

The compression stage processes the input sequence of page ids generated during the rendering process by sorting it using an algorithm based on parallel radix sort. Then a stream compaction step reduces the sorted input sequence by eliminating identical entries. At the same time, the number of

encountered unique entries is counted and stored in an associated sequence. The result of this stream-compaction operation are two sequences storing the requested page ids and the number of their occurrences in the original feedback lists. We call this number the *page coverage* measured in the number of covered feedback pixels. Listing 3.4 illustrates the very compact implementation of this approach based on the Thrust library. The input sequence to this algorithm is the complete payload buffer of the per-pixel feedback lists. For this algorithm to work, the payload buffer must be cleared to a certain value representing an invalid data page. This is done once during system initialization and then after each invocation of the feedback evaluation method.

The two executed sorting and stream-compaction phases for the histogram compression are typically implemented using parallel scan algorithms. The implementation of parallel scan algorithms on modern GPUs possess linear time complexity depending on the size of the input sequence [HSO07]. Therefore, our feedback compression approach is directly dependent on the size of the allocated feedback payload buffer representing the input sequence.

As the histogram compression method is performed using a GPU-compute API and the feedback list generation is performed during rendering with a graphics API, certain buffer resources are shared in both operation contexts. While the compute phase requires shared access to the feedback-list payload buffers, the temporary memory resources required during the compaction phase are pre-allocated in GPU memory accessible only to the compute kernels. The results of the compaction process are finally transferred to host memory for use in the hierarchy-cut selection mechanism.

Page-Coverage Prioritization

The generated list of required data pages and their associated page coverage is used on the CPU for the update of the multi-resolution hierarchy cuts. The employed split/merge algorithm requires a monotonic prioritization function (cf. Section 3.5). This means that each node in the respective multi-resolution hierarchies is assigned a priority with the parent node's priority values larger or equal to the priorities of their child nodes. The priority function in our approach uses the page coverages generated during the feedback evaluation on the GPU as the priority of the particular node.

The requested data pages represent specific nodes in the multi-resolution hierarchies of the large data sets handled by our system. The basic idea to create a monotonous priority function from the available page coverage values associated with each data page is:

1. Assign the page coverages of all data pages to the associated hierarchy nodes as their priority.
2. Propagate the priorities upwards to all parent nodes, accumulating the coverage values.
3. Finally, propagate normalized page coverages downwards to child nodes with yet unassigned priorities to facilitate data pre-fetching.

Through the upwards propagation, we create the basic monotony property of the priority function. Page-coverage values of actually requested data pages generated through the feedback mechanism are represented by integer values larger than or equal to one. The priority down-propagation equally splits the priority of nodes with an assigned coverage to their children and normalizes the thereby created coverage value to the range (0.0, 1.0) with respect to the viewport resolution. Therefore, the monotony property of the priority function is maintained and the data page pre-fetching property is created as described in Section 3.5.2.

For the purpose of storing the acquired coverage information and allowing fast queries of node priorities for arbitrary tree nodes, we employ sparse hierarchical data structures - the so called *priority trees*. Each data set, handled by our data-virtualization system, is assigned such a priority-tree structure, exclusively providing its node's current priorities. For two respectively three-dimensional data sets, these data structures are based on sparse quadtree or octree representations, mirroring the structure of the multi-resolution data representations of the original data sets. These priority-tree structures, however, only contain information about actually requested data blocks leaving out invisible nodes or nodes of higher resolution than actually required. As a result, they depict a sparse tree-hierarchy, representing a particular tree-cut hierarchy of the requested data pages.

Upon receiving an updated compressed feedback list from the level-of-detail evaluation on the GPU, the priority trees are filled with the page coverages of the requested data pages. The individual priority structures associated with the data sets are selected through the data-set instance ids contained in the particular page ids. The page coverages associated with

a requested data page are inserted at the same node position as the node data page represents in its multi-resolution hierarchy. During this insertion process, the page coverages are accumulated in all nodes along the path from the tree's root node to the particular node.

A query for a node priority works by just traversing this data structure and returning the stored page coverage. However, if a node is not part of the sparse tree structure because it is of higher resolution than an actually requested data page, the page coverage of the last actually existing node on the virtual path from the root node is propagated down to the queried node, as required by our priority function design. As the sparse priority-tree data structures only contain information about actually requested data blocks, they exhibit a small memory footprint and are very quickly constructed at run-time.

Summary

In this section we presented a level-of-detail feedback mechanism based on per-pixel linked lists. The lists store a variable number of level-of-detail requests of data pages determined to be required directly during a rendering process. We detailed a GPU-based implementation approach for the concurrent construction of singly linked lists on a per-pixel basis utilizing a paged list layout for efficient insertion operations during run-time. The actually required data pages are determined based on a texture-space metric and are represented in the feedback lists using unique page identifiers, independent of the underlying data set type (e. g. height field or volume). By employing a feedback-evaluation stage directly on the GPU, we create a compressed feedback representation of data pages and associated page coverage in screen-space that is finally transferred to the CPU for use in the multi-resolution hierarchy cut update process.

3.7 Out-of-Core Data Management

After describing the most important parts of our approach to a multi-resolution texture virtualization system in detail, this section now presents the out-of-core data handling of our data-virtualization system on a more abstract concurrent process level. On this level, the system is concerned with several concurrently running tasks including the update of the multi-

resolution hierarchy cuts, the fetching of data pages from external storage, the transfer of data pages constituting working sets to the GPU and, of course, the actual rendering process of visualization applications utilizing our data-virtualization system. In the following sub-sections we will describe our concurrent system design and present details on asynchronous data transfers to the CPU and GPU.

3.7.1 Parallel System Design

The resource management of our out-of-core data-virtualization system is based on the idea of a two-level cache hierarchy. This hierarchy is defined by two large resources: the page cache in system memory acting as a second-level cache for page data loaded from external sources and the page atlas in graphics memory as the first-level cache of page data used during rendering (cf. Section 3.2). The maintenance of these resources as well as the actual data usage during rendering is divided into multiple concurrently running tasks in our system design. The two major processing resources applied in current computer systems include: the CPU on the host system with multiple processing cores and the GPU on the graphics device with dedicated memory connected to the host through the system bus (cf. Section 2.3.1). The graphics device and the host system represent two inherently concurrent processing components. Figure 3.22 schematically illustrates the parallel system processes executed on the CPU and the GPU. The CPU handles asynchronous fetching and decoding of compressed data

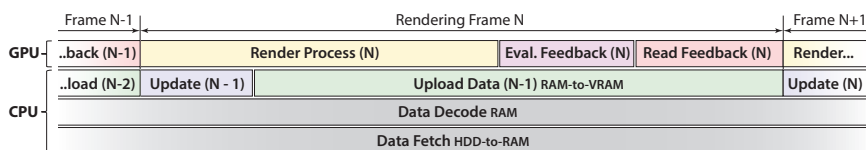


Figure 3.22: This figure shows a single rendering frame and the main system processes running concurrently on the CPU and GPU. The data fetch and decode processes load data pages from external sources to the page cache. Level-of-detail feedback information from the rendering process is evaluated directly on the GPU. A compressed list of required data pages is used by the update process to determine missing data for asynchronous upload to the GPU.

pages from external storage in parallel to the main process driving the update of the multi-resolution hierarchy cuts and the data upload to the graphics memory. The GPU executes the actual rendering approaches while generating the level-of-detail feedback lists. These lists are evaluated and compressed directly following the rendering and transferred to the CPU for the hierarchy cut updates.

3.7.2 Out-of-Core Data Fetching and Caching

The first concurrent process in our system is the fetching of data pages from external storage. Our system employs a single data fetching process for this purpose, implementing an on-demand fetching mechanism using a LRU - least recently used - cache replacement strategy. This process is realized as an asynchronously executed thread on the CPU running mostly independent from all other system tasks. This thread is driven by a fetch request queue. This queue is filled by the unified hierarchy-cut update mechanism. The employed split/merge algorithm, as described in Section 3.5.1, places fetch requests into the request queue when encountering nodes which are not resident in the respective page cache in main memory. The fetching process lies dormant when the queue is empty and is signaled upon pending fetch requests. It then proceeds to load the requested data pages into the respective page cache. The data loading is not bound to any synchronization with the main process despite thread-safe fetch-request insert operations and the respective page-cache updates.

The fetch process is extended by additional concurrently running tasks upon handling compressed page data on external storage (cf. Section 3.3.3). As the data-page caches in our system store the uncompressed page data for the direct upload to graphics memory, the compressed data is decoded directly after loading. Taking advantage of the multi-core architectures of modern computer systems this task can be accelerated by employing multiple concurrently running threads, organized in a thread pool. This way the potential run-time overhead of processing complex compression techniques can be absorbed for a certain number of data pages in order to prevent stalling of the data-page fetching process. The compressed data is fetched as described previously but is not directly stored in the page cache. It is stored in a temporary storage holding a certain number of compressed data pages. This temporary store is accessed by the decoding thread pool. The particular thread scheduled to process a certain compressed data page

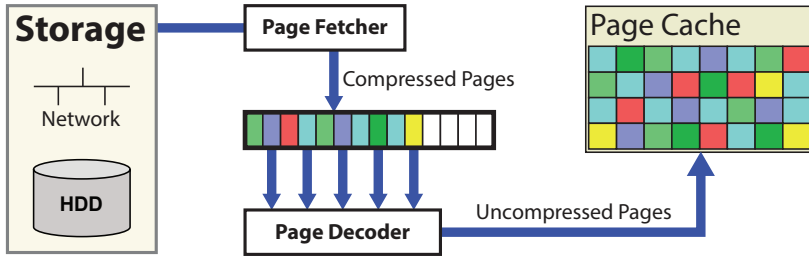


Figure 3.23: This figure shows the process of loading compressed data pages from external storage. The compressed pages are fetched into a temporary storage buffer. Multiple decoding tasks decompress data pages in parallel and finally place the uncompressed data in the associated page cache in main memory.

eventually places the uncompressed page data in the associated page cache. This process is illustrated in Figure 3.23. This approach can absorb longer decoding times for a certain number of compressed pages before stalling the fetching process due to saturating the temporary page storage. While we employ a LRU strategy to replace pages in the page caches, the paging of data pages in the atlas textures is controlled by the update method of the multi-resolution data representations. Data pages currently part of a graphical working set are marked as locked in the page cache and do not take part in the LRU replacement.

We opted to implement a custom data-page caching mechanism instead of relying on the operating system provided automatic paging mechanisms. Application-controlled on-demand paging schemes offer more control over the loading of data pages and, very importantly, over the pruning of no longer required pages as described by Cox et al. [CE97]. The operating system provided mechanism globally manages virtual memory for multiple processes. As a result, a different process can cause the elimination of active data pages from memory and therefore costly reacquisition upon the next data access. We therefore employ custom file handling methods forgoing operating system-level caching of file read and write operations. Furthermore, we control the role of our custom caches in host memory in the operating system’s virtual memory management scheme in order to prevent undesired paging of the associated memory regions. This way we

have full control over the cache sizes and active data pages as well as their potential removal from memory.

3.7.3 Asynchronous Data Updates and Rendering

The next important running system processes are the actual rendering process with the subsequent feedback compression stage running on the GPU and the main system process running on the CPU which is concerned with the actual data management. The GPU and the CPU operate concurrently but the operation of the GPU is controlled from the CPU. The main process performs the following tasks during a rendering frame:

- ▶ Issue the rendering commands, which begins the rendering process on the GPU.
- ▶ Invoke the hierarchy-cut update mechanism using feedback information generated in prior rendering frames.
- ▶ Upload required data to GPU memory and update the atlas textures accordingly.
- ▶ Enqueue the start of the feedback compression stage following the rendering process.
- ▶ Receive compressed level-of-detail feedback list from GPU.

Certain explicit synchronization points exist between the operation of CPU and the GPU, such as the swapping of the front and back buffer. However, several implicit synchronization points are introduced through certain operations, such as data uploads from host to GPU memory. Modern graphics APIs such as OpenGL offer ways to asynchronously transfer data to and from the GPU. We try to employ asynchronous data transfers throughout our data management system. However, particular implicit synchronizations occur due to drawbacks of current OpenGL implementations and design decisions pertaining to our system. Therefore, the parallel design illustrated in Figure 3.22 represents an ideal concurrent concept of our system.

The first overlapped execution on the GPU and the CPU is the rendering process and the update of the multi-resolution hierarchies. Immediately after invoking the rendering process, the CPU regains control while the GPU is processing the visualization. The CPU then proceeds to update the multi-resolution hierarchies using feedback information of a rendering frame

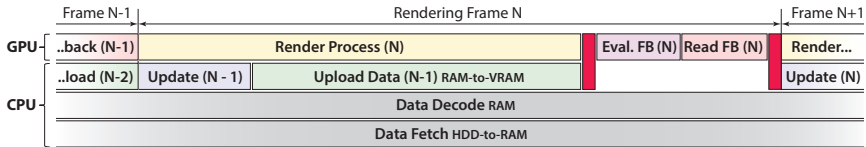


Figure 3.24: This figure shows a more realistic representation of the parallel system processes. The feedback evaluation is performed using a CUDA kernel, which causes context switches from and to OpenGL. This introduces two additional hard synchronization points in our system (represented by red box).

processed earlier. A list of data pages is generated representing data required to be uploaded to the GPU to consistently represent the selected working sets in graphics memory. Directly following this, we start the upload of the particular data pages to GPU memory. As the associated page atlas textures are potentially still in use by the rendering process, we employ dedicated buffer resources on the GPU for the asynchronous upload of new data coincident with the rendering process [Nvi10]. After finishing the upload and the rendering process, the data is copied from this buffer resource to the particular atlas textures using fast on-device memory transfers. This results in an at least two-frame delay between the detection of a missing data page in the working set and its availability to the renderer.

Following the rendering process, the feedback evaluation process is engaged compressing the prior generated level-of-detail feedback information. For this stage, we use a compute kernel executed on the GPU directly after finishing the rendering process (cf. Section 3.6.4). However, due to problems regarding the interoperability of available graphics and compute APIs, this context change introduces a static synchronization point into our system. More specifically, on current generations of GPUs no simultaneous graphics and compute operation is supported, causing hard context switches when launching and upon finishing the histogram-compression stage. These context switches cause the graphics pipeline to finish all previously issued operations (e. g. rendering and data transport). Consequently, Figure 3.24 illustrates a more realistic parallel run-time structure of our system including two hard synchronization points before and after the feedback evaluation.

3.8 Practical Results

In this section we will discuss experiences and practical results of a prototypical implementation of the presented data-virtualization system. We implemented the described system using C++, OpenGL4 and NVIDIA CUDA. The rendering related aspects are based upon OpenGL4 and GLSL, while the GPU-computing stage of the level-of-detail feedback evaluation process is based on CUDA and especially the Thrust parallel algorithms library [BH11]. All tests are performed on a 2.8 GHz Intel Core i7 workstation with 8 GiB RAM equipped with a single NVIDIA GeForce GTX 680 graphics board running Windows 7 x64.

3.8.1 Data-Page Size Ramifications

A fundamental decision for the usage of large data sets within our out-of-core data-virtualization system is the choice of specific data-block dimensions for the initial subdivision and subsequent construction of the multi-resolution hierarchies of the two and three-dimensional data sets. This choice depends on multiple factors: the achievable transfer rates of data pages constituting the data sub-blocks from external storage; the achievable sub-texture update rates of the atlas textures; and the introduced overhead relating to data structure sizes and run-time effects on the actual rendering.

Transfer-Performance from External Storage

The pre-processed multi-resolution hierarchies of the individual data sets are typically stored either on local drives of the visualization computer system or on remote drives connected through a network interface. The main factors for loading data pages from external storage are the achievable transfer rates and the data-access latencies. The performance of data transfers heavily depends on the granularity of the data transfer requests. As the memory size of the data pages, used throughout our out-of-core data management system, define the size of requested data blocks requested and loaded from the external storage, we investigated the transfer performance of varying data-block sizes. We employed two types of currently available hard drives: a classical hard disk drive (HDD) and more a recent solid state drive (SSD). The classical magnetic hard drives offer extremely large storage capacities of currently up to three terabytes per disk. They, however, suffer from large

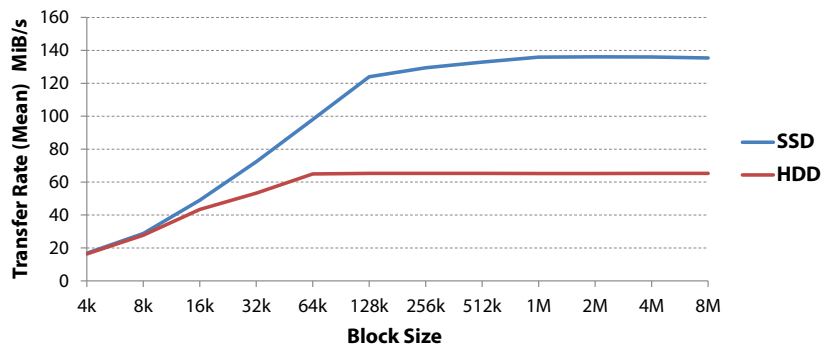


Figure 3.25: Comparison of the achievable read-transfer rates of a classical magnetic hard disk drive (HDD) and a more recent solid state drive (SSD) under different data-transfer block sizes.

data access latencies due to their mechanical nature. Solid state drives on the other hand forgo mechanical parts and store the data persistently using integrated circuits. They therefore offer the best performance in terms of latency and achievable transfer throughput, but are currently only available in much smaller sizes at much higher costs as compared to magnetic drives. Data servers storing massive geological data sets therefore mostly rely on large arrays of magnetic hard drives in RAID configurations for increased data security.

Figure 3.25 shows the results for random data transfers of different block sizes ranging from 4KiB to 8MiB. These tests are performed using a Seagate 7200rpm hard drive and a OCZ Vertex 2 solid state drive connected to the same SATAII controller and formatted with the NTFS file system with a standard block size of 4 KiB. While the magnetic hard drive shows a mean access time of 13.5 ms, the solid state drive reaches access times lower than 0.33 ms. However, when regarding the measured transfer rates, both show comparable characteristics depending on the chosen block sizes. The chart clearly shows the inefficiency of the data transport when using very small data-page sizes. Starting with data-block sizes from 64 KiB to 128 KiB both types of drives begin to produce their maximum transfer rates.

As a result, the data pages stored on the external storage should be larger than 128 KiB. This means that volumetric data sets using a typical 8 bit

voxel representation should be handled with a data block size of at least 64^3 , resulting in 256 KiB data-page memory size. Similarly, two-dimensional height-field data sets using 16 bit texel resolutions should be handled with a block size of at least 256^2 , resulting in 128 KiB data-page memory size (cf. Tables 3.1 and 3.2 on page 61). However, the application of compression methods on the externally stored data pages decreases the actual page size and larger data pages are necessary for best transfer performance as a result.

In regards to data access latency, the chosen z-order layout of our multi-resolution hierarchy file storage (cf. Section 3.3.3) reduces the effects of large data access latencies on magnetic hard drives. Through the orderly traversal of the current tree-cut hierarchies by the selection algorithm, adjacent nodes are potentially accessed together. For example, a node-split operation requests the loading of four respectively eight adjacent nodes in the hierarchy. Thereby, large seek times for loading multiple pages can be avoided in these cases as the required nodes are stored in close proximity to each other. The transfer performance of HDDs, of course, strongly depends on the drive's fragmentation level.

GPU-Data Update Performance

The update of the working sets stored in GPU-bound texture memory is performed in two stages in our system. We first upload new data pages pooled together in a pixel buffer resource (PBO). Then this buffer is used to update the particular page atlas textures directly on the GPU (cf. Section 3.7.3). Therefore, two update rates are of interest for the performance of the out-of-core data management system: the upload-rate of data from the host's main memory to the PBO, and the sub-texture update rate on the GPU.

Table 3.3 shows the transfer rates from the host system's main memory to the GPU as well as the achieved texture update rate when initializing an entire 3D-texture atlas with a single update call. The upload rate through the system bus to the PBO is independent of the actual transferred data type. The achieved transfer rate falls roughly within the range of 50-60% of the theoretically possible bandwidth of the utilized PCIe 2.0 x16 system bus. While we observe quite consistent transfer rates from main memory to the PBO on the graphics device, the achievable memory-transfer rates of the on-device texture updates heavily fluctuate. The utilized GPU offers

Source Memory	Destination Memory	Transfer Rate (GiB/s)
main memory	PBO	4.35
PBO	texture	24.29

Table 3.3: Transfer rates for memory transfers from the host's main memory to a pixel buffer resource (PBO) in GPU-bound memory and for the initialization of a 3D-texture resource using on-device memory transfers. The used 3D-texture employed an 8 bit single channel format and its size was adjusted from 128^3 to 1024^3 incrementally to determine the mean transfer rate.

a theoretical on-device memory bandwidth of 192 GiB/s and the achieved results utilize only a fraction of this value.

The on-device texture data transfer rate shown in Table 3.3 is generated using a single update call to initialize the atlas texture with the contents of the buffer. However, the real-world use case of texture updates in our data management system is to employ a particular number of small sub-texture updates. With this procedure, we observe a lower texture update performance for two-dimensional as well as three-dimensional textures compared to the texture initialization performance. We determined the initialization performance as an upper bound to the sub-texture update operations in our system. The observed results heavily depend on the employed two and three-dimensional data-block sizes as well as the physical resolution of the atlas textures. We found that sub-texture updates on larger atlas textures are less efficient for smaller data-page sizes. This effect is much more pronounced on 3D-textures than on 2D-textures. The cause of this behavior are the proprietary internal texture layouts of current GPUs. The data we upload to the GPU essentially employs a linear memory layout. Textures, however, use specialized cache-coherent layouts to maintain the locality of data samples for efficient access to adjacent samples for fast texture-filtering operations. Upon the sub-texture updates, the linear memory layout uploaded to the PBO needs to be transformed into the internal texture memory layout. These layout changes are completely transparent to developers at the expense of data-transfer performance.

Our data management system offers the ability to adapt to a particular bandwidth budget for the upload of incremental updates to the working sets

of data pages in GPU memory. The size of this bandwidth budget depends on two variables: the actual achievable transfer rates to the GPU and the desired rendering frame times. However, a naïve allocation of the upload budget, depending on the desired visualization update rate, will result in degraded performance. Factors such as fixed overhead to send rendering commands, the hierarchy-cut selection process and the feedback evaluation stage on the GPU influence the actually available portion of the frame time available to data uploads. With the observed increased overhead for the sub-texture updates, the data upload itself represents a considerable portion of the frame time.

In order to maintain a steady frame rate through slow and fast viewer position changes, we employ a relatively small GPU-upload budget. Slow and smooth viewer movements result in only small changes of the working set, while big movements require large updates. In our experience, a small upload budget of under 1 GiB per second facilitates efficient updates of small working set changes and keeps the update overhead minimal for the larger working set adaptations. The limiting factor at this point are the sub-texture updates, as the data upload to the GPU is performed asynchronously to the rendering. With smaller upload budgets, large working set changes are performed over a series of rendering frames amortizing the cost of the sub-texture update operations.

GPU-Upload Consideration

The traditional approach of using dedicated buffer objects on the GPU to upload data to graphics memory only allows for CPU-asynchronous transfers. This means the CPU does not wait for the completion of initiated memory transfers and is freed up for other tasks. The actual memory transport, however, is then typically handled in sequence with the rendering by the GPU. In order to achieve truly asynchronous data transfers to the GPU while rendering modern NVIDIA GPUs offer additional DMA-engines (direct memory access) [Nvi10]. These additional DMA-engines facilitate fully asynchronous memory transports on the GPU. They require the handling of multiple processing threads with individually attached rendering contexts dedicated to distinct processing resources on the GPU. The run-time operation of the DMA-engines is controlled through complex inter-thread and inter-context synchronization. Upon experimenting with data transfers asynchronous to the rendering process using the DMA-engines,

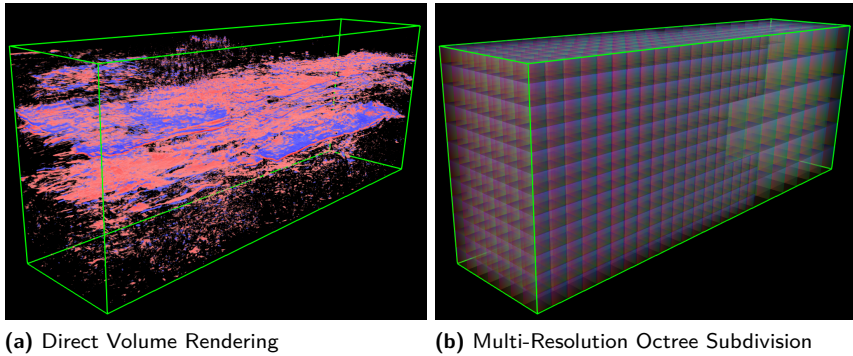


Figure 3.26: Setup of rendering test for the examination of the influence of varying volume brick sizes on rendering performance.

we observed only minimal advantages at the cost of a much more complicated and error-prone process model.

Run-Time Overhead

The choice of the fixed data-page size also influences rendering algorithms based on the serialized tree-cut hierarchies. We examined the influence of varying data-page sizes using a GPU-based direct volume ray casting method employing the serialization of the octree-cut hierarchy of a single volume. The data-page sizes influence the maximum depth of the multi-resolution hierarchies (smaller volume bricks generate deeper octrees), which potentially also affects the granularity of such rendering algorithms. The algorithm used for the evaluation directly traverses the octree data structure and proceeds to sample the contained volume bricks associated with the octree nodes. The used rendering algorithm is explained in more detail in Section 4.3.2 of this thesis. We examined the traversal overhead as well as the achievable rendering performance when using varying data-page sizes.

The presented results are generated using a fixed viewing position and a viewport resolution of 1024×768 . The page-atlas size was 512 MiB visualizing two 8 bit fixed-point precision volumes with the dimensions $1915 \times 439 \times 734$ and $5989 \times 3933 \times 1501$ resulting in raw volume sizes of 588.4 MiB and 32.9 GiB, respectively. In order to rule out effects of the

Brick Size		Octree Size (GiB)	Octree Depth	Draw (ms)	Traverse (ms)	Ratio
Volume 1	32 ³	0.85	6	45.3	3.4	7.5%
	64 ³	0.83	5	36.8	2.8	7.6%
	128 ³	0.84	4	28.5	1.8	6.3%
Volume 2	32 ³	47.2	8	102.4	3.3	3.2%
	64 ³	43.6	7	79.8	2.7	3.4%
	128 ³	41.1	6	75.2	1.7	2.3%

Table 3.4: Relation of draw times and octree traversal overhead for the visualization of virtualized seismic volumes rendered using a direct ray casting approach.

dynamic data-management system, the update of the multi-resolution hierarchy cuts are fixed after allowing them to settle on stable octree subdivisions for the given viewing position. Figure 3.26 illustrates the test setup for the smaller utilized data set and the applied fixed transfer function. We measured the draw times of the complete visualization and the draw times when only traversing the octree hierarchies. The measured draw times are arithmetically averaged over 100 rendering frames.

Table 3.4 shows the measured results of different volume brick sizes for the two tested volumes. The difference in the draw times between the two used volumes is caused by the different visual appearance of the volumes under the same transfer function. The larger volume contained larger translucent regions allowing the viewing rays to much deeper penetrate the data set. From the presented results, it is evident that the visualization is more efficient with larger volume-brick sizes and the traversal overhead is reduced as expected. However, the ratio of the draw times to the octree traversal times is quite consistent. This means that the rendering time increases at the same rate as the traversal overhead of the octree hierarchy. This observation indicates reduced texture-cache performance for smaller bricks. As a single viewing ray traversing the volume hierarchy transitions from one node to the next, the physical texture coordinates of the associated

volume block in the page-atlas texture leap to a different location. Therefore, when first entering a new sub-block of the atlas texture cache misses are generated. As a result, the first lookups into new nodes are more inefficient. With larger nodes, this texture cache thrashing is less evident as fewer node transitions occur along a viewing ray.

Another important factor regarding how the choice of the data-page size influences our virtualization system is the amount of information the level-of-detail feedback mechanism is required to store and process. Smaller data sub-block sizes, especially for virtual volumes, result in an increased number of data-page requests per viewing ray traversed through the virtualized data set. Therefore, we need to allocate larger resources on the GPU to hold the per-pixel feedback information. Consequently, the evaluation of the contained information in these resources through the histogram-compression method increases the computational overhead, which is investigated in the following Section 3.8.2.

Conclusions

Considering the presented influencing factors of the data-page sizes on certain aspects of a rendering system based on our out-of-core data-virtualization system, we come to the following conclusions. For the virtualization of large volumetric data sets, a block size of at least 64^3 should be used, resulting in 256 KiB data-page memory size. Using a one sample wide shared border among adjacent data blocks, this choice adds 10% memory overhead to the storage of the multi-resolution hierarchies (cf. Section 3.3.2). Larger volume block sizes of 256^3 or larger already result in data-page sizes of more than 16 MiB which are too large to allow for effective GPU-bandwidth usage and data-reduction decisions in our system. The representation of partially occluded regions of the data sets benefit from smaller block sizes facilitating more fine grained level-of-detail control. We therefore employ volume data-page sizes of 64^3 and 128^3 depending on the application area. This choice represents a trade between data overhead, the achievable data-transfer rates from external storage, rendering efficiency and the granularity of data management decisions during system run-time.

Similarly, we employ data-block sizes of 256^2 or 512^2 for the handling of two-dimensional height-field representations. While the memory sizes of the resulting data pages are quite small and therefore reduce the efficiency of the loading operations from external storage, larger two-dimensional data

blocks quickly become too big in screen space to allow for fine grained control over the selected local levels of detail.

3.8.2 Per-Pixel Feedback Lists

An integral part of our data-virtualization system is the level-of-detail feedback mechanism, which allows us to directly gather information about required data during rendering. This mechanism essentially consists of two parts: the feedback lists generated for a sub-set of the pixels of the current viewport and the evaluation mechanism compressing the generated information directly on the GPU before transferring the information back to the host system for the multi-resolution hierarchy cut updates. Therefore, relevant factors on the system performance include the insertion rate of feedback requests into the lists and the processing overhead generated through the histogram compression on the GPU.

Fragment-List Insertion Performance

In order to assess the performance of data-request insertions into the feedback lists, we conducted an artificial rendering test. This test renders a variable number of viewport-aligned planes completely covering the screen. We investigated the influence of different list-page sizes, subdividing the payload buffers in our implementation. We completely disabled any data-management facilities as well as the virtual texture lookups during these tests. The initial test generates a feedback list for each pixel of the used viewport, forgoing the supported viewport sub-sampling (cf. Section 3.6.3). We measure the plain draw times for rendering of the plane geometries with and without the actual list insertions in order to obtain the actual list-insertion rates. The results are presented in Figure 3.27.

We observed insertion rates of more than 2.2 billion requests per second for list-page sizes of at least four entries. The measured results include the level-of-detail estimation as well as the encoding of the determined hierarchy node into our compact page-id representation (cf. Section 3.6.2). The smaller list-page sizes exhibit the lowest insertion performance, because more list pages are required to be allocated, thereby increasing the overall data-management overhead per list insertion operation. On the other hand, the smaller sizes present the lowest memory overhead for lower numbers of insertions into individual lists. We achieved the best performance using

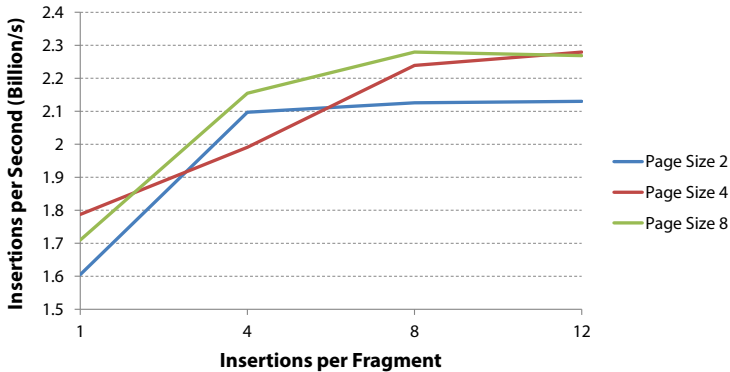


Figure 3.27: Comparison of the fragment list insertion performance under varying list page sizes.

list-page sizes of four and eight elements with identical insertion rates for higher numbers of inserted requests and reasonable memory overhead for small per-pixel lists.

The sub-sampling of the viewport does not increase the overall performance of the list generation process. By only creating a list for a sub-set of viewport pixels, we expected a reduction of the list insertion overhead during rendering. Current GPUs operate on a larger set of pixels in parallel, the so called *pixel quads*³. These pixel quads are assigned to parallel processors on the GPU working in SIMD (single instruction multiple data) order. This means that divergent computations in single pixels of a quad increase the processing times of all processed pixels of the respective quad. We expected using larger sub-sampling factors to skip the list generation for entire pixel quads. However, during our tests we observed a constant per-frame rendering overhead of the list generation, independent of the employed sub-sampling factors ranging from 2×2 up to 8×8 pixels. This means that we achieve only a fraction of the list-insertion performance relative to the chosen sub-sampling factor (e.g. a quarter of the list-insertion performance for 2×2 sub-sampling). Beginning with sub-sampling factors of 16×16 , we were able to produce the expected increase in performance.

³Depending on the GPU generation the pixel quads subdivide the viewport into groups of 2×2 to 2×4 pixels.

From this results, we deduce that multiple pixel quads get grouped together and are processed concurrently in SIMD fashion. With larger sub-sampling factors, we are able to skip the feedback generation and list insertion for larger groups. However, such sub-sampling factors are too large to ensure meaningful feedback and should therefore only be used for extreme viewport resolutions.

The main advantage we gain through the sub-sampling of the viewport for the feedback-list generation is a dramatic reduction of the amount of information generated and stored in the resources associated with the per-pixel lists. Therefore, the memory overhead for the storage of these resources is greatly reduced using the viewport sub-sampling mechanism. As a consequence the execution times for the evaluation process can be minimized using varying sub-sampling factors for differing application scenarios.

Feedback Evaluation Performance

The execution time of the feedback-evaluation and compression process represents a fixed share of the frame times in our system (cf. Section 3.7.1). The implementation of this stage is based on a parallel stream compactification approach on the generated payload buffer containing the actual feedback information of the individual per-pixel feedback lists (cf. Section 3.6.4). The achievable performance is directly dependent on the size of these buffer resources. Through the utilized adaptive allocation scheme, these buffers can grow and shrink to accommodate the requirements for the generated amount of feedback information (e. g. for large translucent volume primitive).

Figure 3.28 shows the achieved evaluation performance of our feedback evaluation implementation. It is evident that the evaluation time grows linearly with the size of the processed feedback buffer. The measured evaluation times quickly surpass 10 ms for larger feedback-buffer resources. The generated feedback buffers for this test, however, are much bigger than the buffers generated during regular system usage. Dependent on the employed viewport size, we vary the utilized viewport sub-sampling factor. For visualizations employing a viewport resolution of up to 1920×1200 , we typically use a 4×4 sub-sampling. The exemplary visualizations shown throughout this thesis typically employed feedback buffer sizes under 1.5 million entries, resulting in evaluation times around 3 ms per frame. In extreme cases, we observed sizes of around 4 to 5 million entries, requiring up to 10 ms for the evaluation. In order to deal with larger viewport resolutions

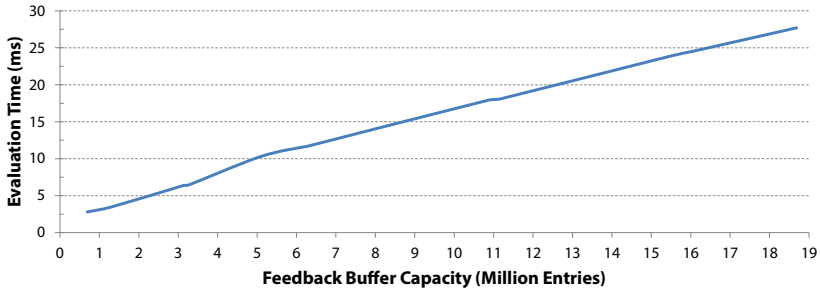


Figure 3.28: Evaluation times for different amounts of generated feedback information.

or visualizations requiring larger feedback resources, we adapt the viewport sub-sampling. Currently a static sub-sampling factor is employed at system run-time. However, in order to facilitate dynamic adaptation to a fixed frame-time budget for the feedback evaluation, this factor and therefore the feedback-buffer resources could be dynamically adjusted at run-time.

Regarding our feedback-evaluation implementation based on a compute stage following the rendering process (cf. Section 3.6.4), we observed particular inter-operation problems of the utilized CUDA and OpenGL APIs. The buffers associated with the feedback lists are OpenGL resources filled during the rendering stage. These buffers are then mapped to and shared with the CUDA compute context, where they are read and finally cleared. Following this, the buffers are unmapped from the compute context to be used again for the next rendering frame from the OpenGL side. We observed unusually long unmap operations for larger shared buffer resources, indicating a hidden memory-copy operation between the two operation contexts. These measured unmap-times vary between virtually non-existent to up to multiples of the actual CUDA kernel run-time. Experiments forgoing write operations to OpenGL buffer resources by only granting read-access for the CUDA kernel did not show any improvements. Exchanging the CUDA implementation with an OpenCL implementation proved to be much more problematic with even larger costs associated with the mapping operations of buffer resources between the two operation contexts. We therefore try to keep the feedback buffers small to minimize this effect. We are expecting future revisions of the OpenGL as well as CUDA implementations to solve

such inter-operation issues. Moreover, with the very recent introduction of OpenGL version 4.3 [SA12], specialized compute shader facilities are added to the rendering pipeline. An implementation of the feedback-evaluation process using such functionality forgoes any inter-operation problems between distinct GPU-graphics and compute APIs.

3.8.3 Bindless Textures

Our data-virtualization system utilizes atlas textures on the GPU to store all data pages constituting working sets, making them jointly accessible to shader programs. This design choice is based on the fact that modern GPUs only allow simultaneous access to a small number of texture resources. The use of atlas textures, however, presents certain drawbacks regarding the update of small sub-texture regions as described in Section 3.8.1.

The most recent generation of NVIDIA GPUs from the GK1xx series offers a new feature called *bindless textures* (available through the OpenGL extension `NV_bindless_texture`). This feature allows to access a virtually unlimited number of individual texture resources from shader programs. This enables us to replace the atlas-texture resources currently employed by our texture-virtualization system with a large set of small textures representing individual data pages.

This approach offers two major advantages. Firstly, the texture updates relinquish the previously mandatory sub-texture updates by updating the small page textures completely. Secondly, the encoding of the data pages in the serialized hierarchy cuts is simplified. With individual textures, we are able to achieve much more stable on-device texture update rates. More interesting, however, is that when using this feature a data page in the texture pool is identified by a single identifier instead of a three-dimensional position in the texture atlas (cf. Section 3.4.2). We are therefore able to reduce the 64 bit node representation to 32 bit representation, halving the memory requirements for the serialized tree-cut hierarchies on the GPU, potentially increasing the cache efficiency of our data structure. Additionally, the virtual texture coordinate translation is simplified, because after generating the page sampling address, the sample is fetched directly from the texture associated with the data page without the additional scale and bias operation into the atlas texture coordinate system.

Despite the simplifications and potential performance enhancements facilitated by the bindless texture feature, the actual rendering performance,

observed in an experimental implementation, suffered an extreme performance drop. While all system components performed at least equally as well as with the atlas texture, the rendering times increased by a factor of three. After investigating this issue more closely we conclude, that the texture cache thrashing issues also apparent with the atlas-texture approach are much more pronounced when using individual data-page textures. The absolute cause of this issue is again hidden in the respective OpenGL implementation. However, personal statements of NVIDIA engineers support our conclusion. Therefore, the usage of this feature theoretically offers major advantages for the implementation of a data-virtualization system, but currently fails to deliver comparable rendering performance to an atlas-texture approach. Future generations of GPUs may allow a more efficient application of bindless textures.

3.9 Summary

In this chapter we presented the design of a unified out-of-core data-virtualization system that is able to simultaneously handle multiple large volume and height-field image resources, occurring in large geological models. We employ multi-resolution data representations respectively based on quadtree or octree data structures to generate level-of-detail working sets of data stored in graphics memory. The working sets are represented in graphics memory by atlas textures, specifically assigned to the two and three-dimensional data types handled by the system. We employ an out-of-core data management based on a two-level data-caching strategy of data loaded from external storage into system and subsequently into graphics memory. While existing out-of-core data management and rendering approaches are mostly specialized for single data primitives, such as large volumetric data sets or large height-field surfaces, the presented out-of-core data management system handles multiple instances of distinct data types simultaneously. Through the mutual utilization of memory resources in main as well as graphics memory by multiple data sets, our data management system is able to share and balance system resources among different data sets. While the sharing of memory resources is restricted to data sets of equal type (e. g. height field or volume), this limitation does not affect the balancing of other important system resources such as limited bandwidth

between the external storage and the host system as well as between the main and the graphics memory.

The working sets of data sub-blocks stored in graphics memory are described by tree-cuts of the associated multi-resolution hierarchies. The required indirection information, to translate virtual sample locations to physical sample coordinates in the particular atlas textures, is provided either through page-index textures or tree-cut serializations. The complexities of locating a particular data block and the virtual coordinate translation into the atlas coordinate system is hidden from potential users of the data sets through an abstraction layer emulating regular texture-data access functionality. However, the texture data-virtualization stage of our system on the GPU supports varying levels of data abstraction for different usage scenarios. While advanced rendering algorithms can access the internally utilized quadtree and octree serializations on the GPU, existing or basic visualizations are able to sample the virtualized large volume and height-field data resources in the same way as regular texture resources.

The working set generation in our system is based on a unified level-of-detail feedback system with inherent support for translucent geometric and volumetric data sets. This feedback mechanism is based on the fundamental idea to determine required data blocks of the employed multi-resolution hierarchies directly during the rendering process. Through the generation of per-pixel linked lists, we facilitate the storage of a varying number of feedback requests for a subset of viewport pixels. The actually required data blocks are determined through a texture-space metric and are represented in the feedback lists using unique page identifiers, independent of the underlying data set type. By employing a feedback-evaluation stage directly on the GPU, we create a compressed feedback representation of required data blocks, which is finally transferred to the CPU for use in the multi-resolution hierarchy cut update process. This feedback system is jointly used for all data sets of both supported data types (e. g. height field or volume) without special treatment of individual primitives. By either exploiting inherent hardware features or by using custom image-order rendering approaches, our system is capable of resolving data visibility without the application of any costly occlusion culling approaches.

In its entirety, our data-virtualization system represents a scalable, demand-driven out-of-core data management system capable of supporting massive seismic models composed of multiple large data sets.

Chapter 4

Rendering of Virtualized Seismic Data Sets

IN THIS CHAPTER we present GPU-based rendering methods for the visualization of large seismic data sets. Existing geo-scientific visualization systems struggle with two factors: the size and the amount of individual components in such data collections. We demonstrate the application of our out-of-core data-virtualization system to this specific application area in order to handle geological models consisting of multiple large seismic volumes as well as large horizon height-field surfaces. In particular, we present a volume ray casting method for rendering multiple arbitrarily overlapping multi-resolution volume data sets in Section 4.1. We show how efficient volume virtualization allows for multi-resolution volumes to be treated exactly the same way as regular volumes, simplifying the process of identifying overlapping and non-overlapping volume regions. In Section 4.2 we present a ray casting-based rendering system for the visualization of geological subsurface models consisting of multiple highly detailed height fields. Finally, in Section 4.3 we present a rendering system for the combined visualization of entire geological models consisting of highly detailed stacked horizon surface geometries and massive volume data. Through the virtualization of the access to each data component and the level-of-detail feedback mechanism, we show how individual volumes or horizon height fields can be treated at locally varying levels of detail, while inherently taking data occlusions between all displayed primitives into account.

4.1 Multi-Volume Ray Casting

In the course of the production of large oil fields, often multiple seismic volumes are generated by either acquiring adjacent areas, in order to survey

larger regions, or by reacquiring the same area at different points in time to track the development of an in-production oil field (cf. Section 1.2.2).

4.1.1 Problem Setting

The individual acquired volumes often exhibit different resolutions, different orientations and can be partially or even fully overlapping. While such multi-volume data sets are typically re-sampled and merged into a single large volume, this process is very undesirable due to the extended pre-processing times, numerical inaccuracies and data bloat. The distinct handling of separate arbitrarily overlapping volumes can vastly improve the flexibility and efficiency of the visualization of multiple seismic surveys.

Specialized multi-volume rendering approaches overcome most of these drawbacks by directly visualizing scenes containing multiple overlapping volume data sets (cf. Section 2.3.4). Key for the efficient, simultaneous visualization of multi-volume scenes is an effective identification of overlapping and non-overlapping volume regions in order to avoid costly oversampling of spatial segments occupied only by single volumes and to enable correct composition of actual multi-volume segments. Cai et al. [CS99] distinguished between image, accumulation and illumination-level volume-intermixing schemes, defining how spatially overlapping volume segments are visually combined.

First proposed approaches for rendering of multi-volume data sets are able to visualize scenes composed of moderately sized volumes completely fitting into the available memory resources [GBAG04, RTF⁺06, RBE08]. However, these methods are not directly applicable to visualization approaches for large multi-resolution volume data sets. Plate et al. [PHF07] demonstrated a GPU-based multi-volume rendering system capable of handling multiple multi-resolution data sets. They identified overlapping volume regions by intersecting the bounding geometries of the individual volumes or utilized volume lenses as well as the individual sub-blocks of the multi-resolution octree hierarchy. As a result, an extremely large number of volume fragments is generated without any inherent depth ordering. However, a strict depth ordering is essential for the correct composition of the rendering results of the single volume fragments. As they still rely on a classic slice-based volume rendering method, the geometry processing overhead quickly becomes the limiting factor when moving either individual volumes or the viewer position. Similar to Rößler et al. [RBE08], costly depth-sorting operations of the

generated volume fragments are applied using complex GPU-based depth peeling techniques.

Design Goals

With our multi-volume rendering method, we are pursuing the following design goals:

- ▶ Interactive visualization of large, arbitrarily oriented and overlapping multi-resolution volumes using a GPU-based volume ray casting approach.
- ▶ Support for dynamic scene manipulations, such as moving individual volume data sets or volume lenses.
- ▶ Real-time decomposition of overlapping volume segments on the bounding geometry level instead of on the individual octree nodes.
- ▶ Minimal overhead for the generation of a strict depth ordering of the resulting volume fragments.

We employ a GPU-based volume ray casting method for the visualization of the individual volume fragments. This approach facilitates a straightforward integration of volume rendering optimizations such as early ray termination. The overlapping and non-overlapping volume regions are identified and sorted in front-to-back order using a hierarchical binary space partitioning (BSP). The main advantage of this approach is that only bounding boxes of the cube-shaped volumes or volume lenses are dealt with in the BSP tree instead of the view-dependent brick partitions of the involved multi-resolution volume representations. As a result, our approach requires recomputations of the BSP tree only if the spatial relationship of the volumes changes.

Targeted GPU Generation

The targeted generation of GPUs, available at the time of the development of the rendering algorithm described in this section, is the first to allow the implementation of a single-pass multi-resolution volume ray casting technique. It, however, does not facilitate the implementation of the more advanced features of our data-virtualization system, particularly the direct level-of-detail feedback mechanism. The proposed rendering algorithm

therefore is based on a view-dependent level-of-detail selection function (Section 3.5.3). Consequently, any kind of data occlusion present in the multi-volume scenes is not considered by the multi-resolution hierarchy selection algorithm.

4.1.2 Ray Casting Virtualized Volumes

The individual large volumes contained in a scene are handled by our out-of-core data-virtualization system. Based on the shared resource management, we balance the memory requirements of all volumes against each other. If, for example, volumes are moved out of the viewing frustum or are less prominent in the current scene, the unused resources can be easily shifted to other volumes without costly reallocation operations in system and graphics memory (cf. Section 3.4.1).

Gobbetti et al. [GMG08] and Crassin et al. [CNLE09] presented first direct multi-resolution volume ray casting approaches for single large volume data sets. They employed the traversal of a compact octree data structure on the GPU to facilitate the locating of the individual volume bricks in an atlas texture. Considering the concurrent traversal of multiple octree hierarchies representing volumes of arbitrary orientation in our application scenario, we chose to base our multi-resolution volume rendering approach on a basic volume ray casting method extended by virtual volume-sample lookups enabled by our texture data-virtualization facilities.

Consequently, this specific application is depending on fast and efficient per-sample volume virtualization. Therefore, we are employing the page-index texture multi-resolution hierarchy cut representation of the individual virtualized volumes (cf. Section 3.4.2). This approach reduces the per-sample logarithmic traversal costs of the more advanced hierarchy-cut serialization scheme to a fixed sample-lookup and coordinate-translation overhead. A texture lookup into a virtualized volume texture requires the following two steps. The first step involves sampling the page-index texture at the requested location which results in an index vector containing indirection information about the scale and position of the corresponding volume brick in the associated page-atlas texture. Using this information, the requested sampling position is then transformed to the atlas texture coordinate system and the respective sample is returned. By using this index texture approach, we exchange fast access to the required atlas-texture indirection information for a moderately larger memory footprint. These index textures are several

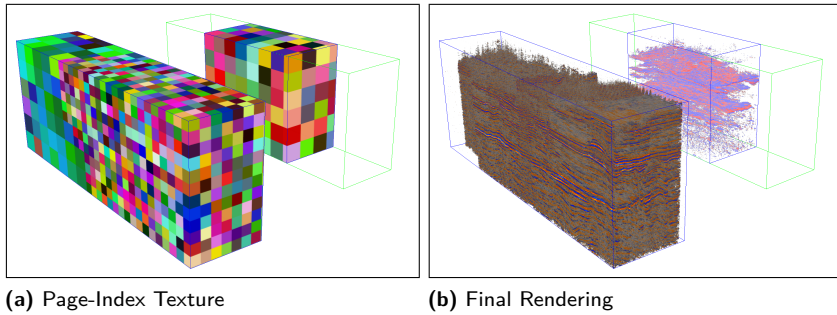


Figure 4.1: Visualization of two spatially separate virtualized volumes. (a) shows the page-index textures associated with the two volumes in the scene. (b) shows the final rendering based on single-pass volume ray casting.

orders of magnitude smaller in size than the actual volumes because they describe the underlying octree representation on a brick level.

Integrating our multi-resolution volume virtualization approach with single-pass volume ray casting is realized as a straightforward extension of the ray traversal. Our approach extends upon single pass volume ray casting methods as demonstrated by Scharsach [Sch05] and Stegmaier et al. [SSKE05], relying on small volumes fitting into graphics memory. Through the completely virtualized volumes, the basic ray casting algorithm remains completely unaware of the underlying octree hierarchy. Only the data-lookup routine is extended, which hides all of the complexities from the rest of the ray casting method. Figure 4.1 shows an example of a scene consisting of two seismic volumes, displaying the underlying index textures using pseudo-colors and the final rendering result using a single rendering pass per volume.

4.1.3 Multi-Volume Decomposition and Rendering

For the visualization of multiple arbitrarily overlapping volume data sets, it is important to differentiate between mono-volume and multi-volume segments as emphasized by Grimm et al. [GBAG04]. They identified different segments along the ray paths for a CPU-based ray casting implementation. In contrast, we segment the overlapping volumes and use a GPU-based ray

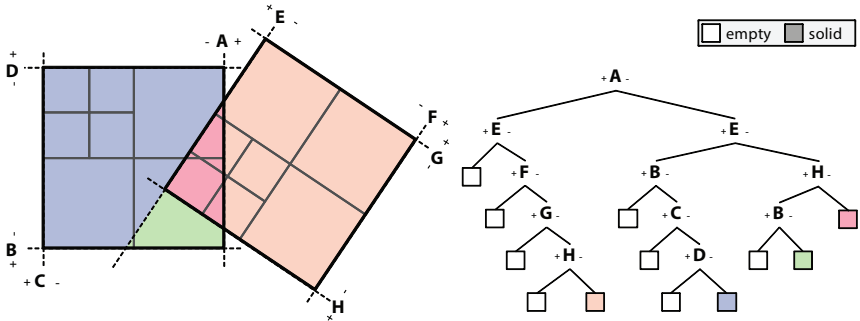


Figure 4.2: Decomposition of two multi-resolution volumes into homogeneous volume fragments using an auto-partitioning solid-leaf BSP-tree. Only the bounding geometries of the volumes are used for the decomposition.

casting approach. We use a BSP-tree-based approach similar to Lindholm et al. [LLHY09] to identify overlapping and non-overlapping volume fragments. While they also supported multi-resolution volume data sets, their approach treats each brick in the multi-resolution hierarchy as a separate sub-volume. Thus, the BSP process generates an immense amount of volume fragments, which need to be rendered in sequential rendering passes. As a consequence, changing the view causes updates of the multi-resolution hierarchy, which forces them to recreate the complex BSP-tree. In contrast, our efficient volume virtualization enables us to treat multi-resolution volumes in exactly the same way as regular volumes by only processing their bounding geometries, which only needs to happen during the initial setup or if the actual volumes or volume lenses are moved in relation to each other.

Solid-Leaf BSP-Tree Decomposition

The BSP construction is based on an auto-partitioning solid-leaf BSP-tree [FKN80]. The BSP-tree is generated using the bounding geometries of the individual volumes or used volume lenses, which also define the split planes. Solid-leaf BSP-trees describe the solid space occupied by the volumes in the scene. The split planes associated with the internal nodes of the tree separate solid from empty space. Due to the fact that the input bounding volumes themselves represent solid geometries, the non-empty leaf nodes

of the resulting BSP-tree describe closed, convex polyhedra containing a fixed set of volumes. Figure 4.2 illustrates a volume decomposition for two multi-resolution volumes creating four convex polyhedra containing only one fragment, which is overlapped by both volumes. The BSP-tree facilitates efficient depth sorting of the resulting volume fragments on the CPU, which is required for correct traversal during the actual volume ray casting process.

Ray Casting of Volume Fragments

The volume ray casting method processes all visible volume fragments in front-to-back order. Each fragment is processed in a single ray casting rendering pass independent of the number of contained bricks. Consequently, the rendering of the decomposed multi-volume scene is implemented as a multi-pass rendering process. During this rendering process, two intermediate buffers are utilized: an integration buffer, storing the intermediate volume rendering integral for all rays, and an auxiliary buffer, storing the ray exit positions of the currently processed volume fragment. We need to generate the exit positions explicitly because the irregular geometry of the volume fragments generated by the BSP-process does not allow straightforward analytical ray-exit computations on the GPU. During the exit-point generation, the accumulated opacity from the integration buffer is copied to this buffer to circumvent potential read-write conflicts during ray casting when writing to the integration buffer. The accumulated opacity is used for early ray termination of individual rays.

For each rendering frame of our multi-volume rendering system, the following steps are performed:

1. Update of BSP-tree volume decomposition in case volumes or volume lenses moved.
2. Generate a strict depth ordering of volume fragments through BSP-tree traversal.
3. Clear the intermediate integration buffer.
4. For each volume fragment in front-to-back order:
 - a) Render volume fragment polyhedron back-faces into second intermediate buffer to generate ray-exit positions.

- b) Render volume fragment polyhedron front-faces to trigger volume ray casting algorithm.
- 5. Compose the contents of the integration buffer to the frame buffer.

During the rendering of an individual volume fragment, the ray-exit positions are read from the second intermediate buffer to eventually terminate the ray traversal upon leaving the particular fragment. This buffer is never required to be cleared. Based on the fact that the volume fragments resulting from the solid-leaf BSP decomposition represent convex polyhedra, the rendering of the polyhedron back-faces for the generation of the ray-exit positions solely affects the viewport area covered by the respective volume fragment. The results of the individual ray casting rendering passes are composed into the integration buffer, incrementally accumulating the complete volume integral for each viewing ray. After all fragments are processed, the content of the integration buffer is blended into the main frame buffer eventually presenting the combined rendering of the overlapping volumes.

The actual ray traversal is based on a global fixed sampling rate. We derive this global sampling rate based on the smallest occurring sample distance in the scene depending on the resolution of the individual volume data sets and their spatial extent in world space. In order to ensure consistent sampling at the borders between adjacent volume fragments, the ray-entry position, generated through the rasterization of the front-faces of the associated polyhedron, is shifted along the particular viewing ray to the next integer multiple of the determined global sampling rate. As a result, the distance between successive data samples generated in two distinct volume fragments is consistent for the entire visualization. Consequently, no additional opacity correction on volume-fragment boundaries is required for correct compositing calculations.

The intermixing of multiple volume data sets in the overlapping regions in our approach is based on the accumulation level scheme described by Cai et al. [CS99]. For the combination of multiple volume data sets based on a single shader program, two options exist: the use of conditionals in the program to sample the correct amount of volumes, and the use of shader instantiation to automatically generate specialized shader programs for certain amounts of volume primitives. While the employed generation of GPUs supports conditional statements in shader programs, it also introduces performance penalties [HB05]. For this reason, we employ shader instantiation to generate specialized ray casting programs for the different

number of volumes overlapping a particular fragment similar to Rößler et al. [RBE08].

4.1.4 Results

We implemented the described rendering system using C++, OpenGL3 and GLSL. The evaluation was performed on a 2.8 GHz Intel Core 2 Quad workstation with 8 GiB RAM and a single NVIDIA GeForce GTX 280 graphics board running Windows XP x64 Edition.

We tested our out-of-core data management system with various large data sets with sizes ranging from 700 MiB up to 40 GiB. The used data sets were seismic volumes from the oil and gas domain. For the evaluation of our multi-volume rendering approach, we used scenes composed of multiple large seismic multi-resolution volumes. Due to limited access to true seismic multi-volume data sets, we conducted the tests using a single data set containing $1915 \times 439 \times 734$ voxels at 8bits/sample. We duplicated the volume several times to convey the ability of our system to interactively handle large amounts of data. The chosen volume-brick size for all volumes was 64^3 , the atlas texture size was 512 MiB and the data cache size in main memory 3 GiB. Images were rendered using a viewport resolution of 1280×720 .

For the evaluation, we emulated potential application scenarios for multi-volume rendering techniques. Seismic models of large oil fields contain multiple potentially overlapping seismic surveys, which can only be inspected one at a time or adjacent surveys have to be merged. Using our multi-resolution multi-volume rendering approach, arbitrary configurations can be directly rendered without data-size limitations, the need for re-sampling or adjacency relationships. Figures 4.3 to 4.5 show some artificial configurations of sets of seismic surveys.

Our system is able to handle many large volumes simultaneously through the shared resource management for multiple multi-resolution volumes. Figure 4.5 shows a scene composed of nine multi-resolution volumes managed through the shared data cache and atlas texture employed by our data-virtualization system. The rendering performance using our multi-volume ray casting system for virtualized multi-resolution volumes is mainly dependent on the screen projection size of the volumes, the used volume sampling rate and the chosen transfer functions. The memory transfers between the shared resources have only little influence on the rendering performance.

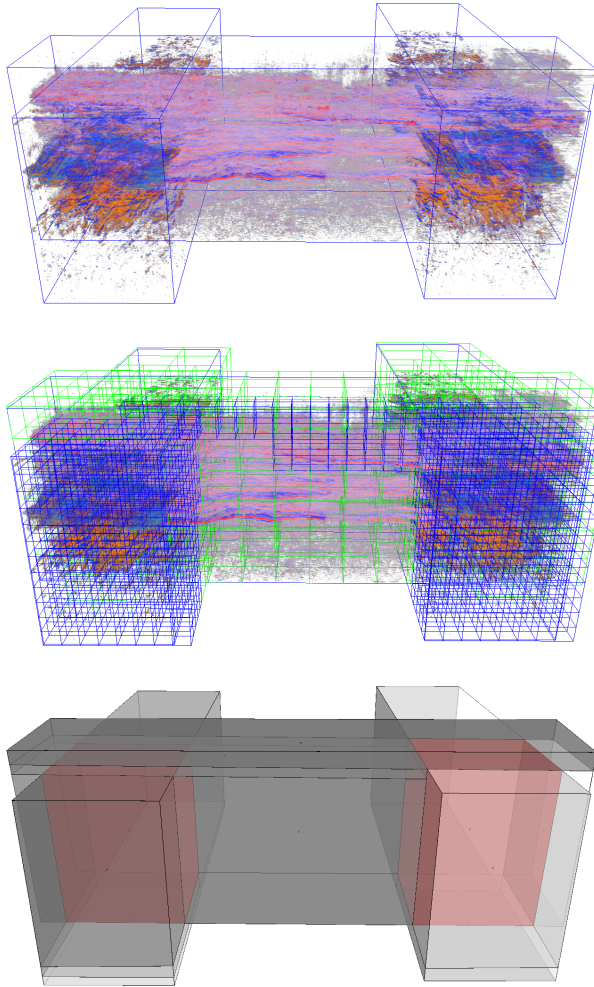


Figure 4.3: Example scenario 1: Three volumes with at most two overlapping volumes. The images show the final rendering, the selected octree cuts and the current volume BSP-tree decomposition. The volume-fragment depth ordering is illustrated through the brightness and multi-volume fragments are shown in red.

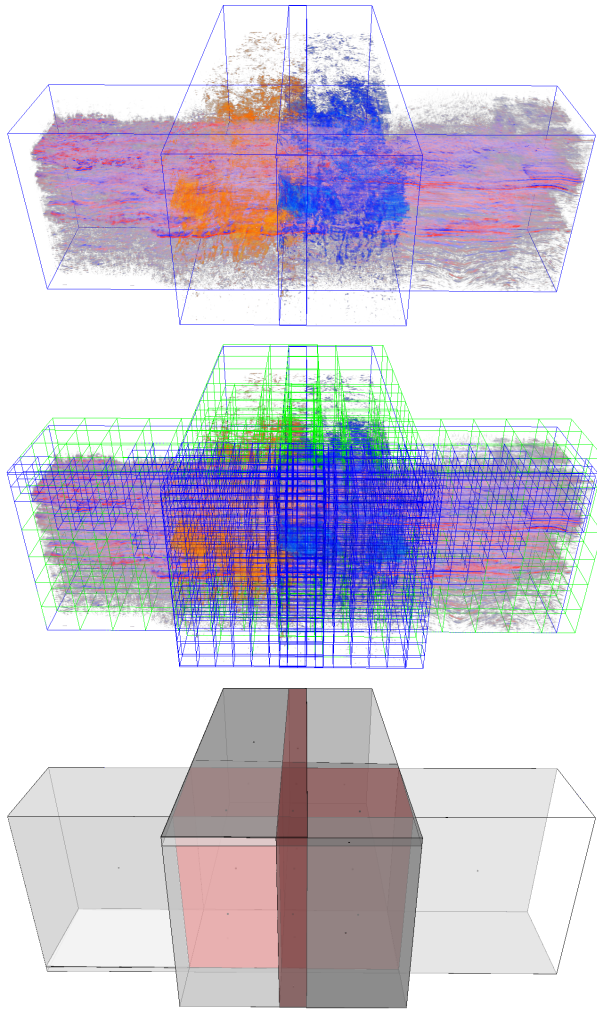


Figure 4.4: Example scenario 2: Three volumes with at most three overlapping volumes. The images show the final rendering, the selected octree cuts and the current volume BSP-tree decomposition. The volume-fragment depth ordering is illustrated through the brightness and multi-volume fragments are shown in red.

Example Scenario	Volumes	Overlapping Volumes	Volume Fragments	BSP-Tree Update Time	Rendering Frame Rate
1	3	2	13	0.15 ms	22 Hz
2	3	3	44	0.28 ms	16 Hz
3	9	0	9	0.29 ms	14 Hz

Table 4.1: BSP-tree update and frame rendering times for the example scenarios shown in Figures 4.3 to 4.5. The brick size for all volumes is 64^3 , the atlas texture size is 512 MiB and the brick data cache size is 3 GiB using a viewport resolution of 1280×720 .

Table 4.1 shows the achieved BSP-tree-update times and the rendering frame rates for the example scenarios shown in Figures 4.3 to 4.5. The observed frame rates for these three scenarios ranged from 12 to 30 Hz depending on the viewer-position. We also experimented with artificial scenarios containing up to nine overlapping volumes. The number of volume fragments grew quickly to 500 fragments requiring BSP-update times up to several milliseconds, which resulted in a large number of required rendering passes. While the BSP-update affects rendering performance during volume manipulation, viewer navigation remained fluent. For the scenario shown in Figure 4.5 containing nine non-overlapping volumes, we also compared the performance of our multi-volume rendering approach to a single-volume rendering implementation, which renders and blends non-overlapping volumes sequentially. We observed frame rates very similar to our multi-volume approach. Thus, the overhead for maintaining the multiple render targets for the ray casting method in such a scenario is small compared to the actual cost for the ray traversal and volume sampling.

4.1.5 Conclusions

We presented a GPU-based volume ray casting system for multiple arbitrarily overlapping multi-resolution volume data sets. The system is able to simultaneously handle a large number of multi-gigabyte volumes through a shared resource management system. We differentiate overlapping from non-overlapping volume regions by using a BSP-tree based method, which additionally provides us with a straightforward depth ordering of the result-

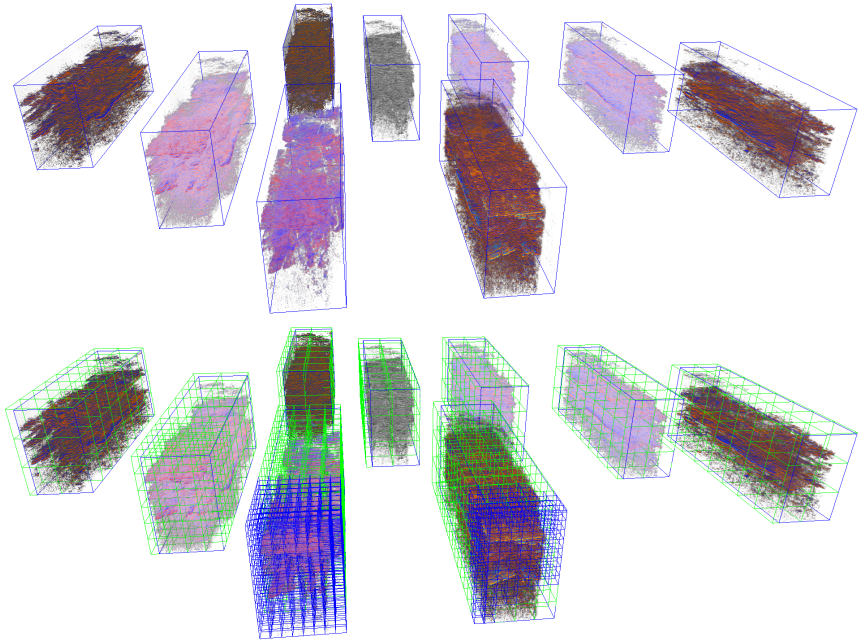


Figure 4.5: Example scenario 3: Nine separate volumes. The images show the final rendering and the selected octree cuts of the displayed multi-resolution volumes.

ing convex volume fragments and avoids costly depth peeling procedures. The resulting volume fragments are efficiently rendered by custom instantiated shader programs. Through our efficient volume virtualization method, we are able to solely base the BSP volume decomposition on the bounding geometries of the volumes or utilized volume lenses. As a result, the BSP needs to be updated only if individual volumes are moved in contrast to the similar multi-resolution method as proposed by Lindholm et al. [LLHY09].

The generation of the multi-resolution representations for this rendering approach is based on view-dependent criteria only. Approaches taking data occlusion information into account can greatly improve the volume refinement process and the guidance of the resource distribution among the volumes in the scene. Furthermore, introducing additional and user-

definable composition modes for the combination of overlapping volumes can increase visual expressiveness [CS99, PHF07].

4.2 Ray Casting Stacked Horizons

Seismic volumes are a central entity in large seismic data collections. They are recorded through elaborate sound propagation procedures, resulting in large volume data sets representing the magnitude of seismic wave-reflections in the earth's subsurface. These seismic volumes represent the origin of most other components of seismic data sets. Geologists and geophysicists use these volumes to create a geological model of the most important subsurface structures in order to identify potential hydrocarbon reservoirs. Of all the components of the geological models, horizon surfaces are one of the most fundamental parts representing the interface between layers of different materials in the ground (cf. Section 1.1). Typical geological models contain multiple horizon surfaces stacked on top of one another. These surfaces generally exhibit a height field like topology.

4.2.1 Problem Setting

The ever increasing size of seismic surveys generates extremely large horizon geometries composed of hundreds of millions of points. The layered character of these data sets leads to high depth complexities in the models due to the mutually occluding and partially overlapping horizon surfaces. In order to gain an overview of such complex stacked models, users often employ translucent display of individual horizon surfaces as illustrated in Figure 4.6.

While horizon surfaces are commonly represented as height fields, general triangulation-based continuous level-of-detail terrain rendering approaches have not been adapted to deal with the specific structure and requirements of multiple horizon layers [PG07]. Furthermore, with increasing screen resolutions and screen-space errors below one pixel, the geometry throughput of current GPUs is becoming a major performance bottleneck and ray casting techniques can provide comparable performance for large data sets [DSW09, DKW09].

Through the rapid evolution of programmable graphics hardware, texture-based ray casting methods implemented directly on the GPU were introduced [OBM00, POC05, PO06]. These first approaches determine the inter-

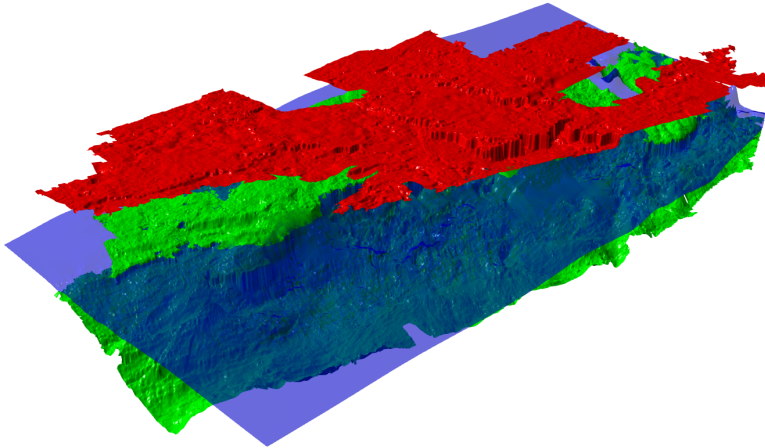


Figure 4.6: This figure shows three different horizons extracted from a common seismic survey. While the upper most horizon (red) is spatially isolated, the lower horizons (green and blue) are partially overlapping. The original resolution of each horizon is 6889×16984 points.

section between a viewing ray and the height field based on a combination of a uniform linear search and a successive binary hit-point refinement. The elevation maps are stored as 2D-texture resources in graphics memory and are thereby limited by the maximum supported texture dimensions. Even for elevation maps of constrained size, these approaches quickly become inefficient for real-time application due to the large overhead of the uniform linear searches. Later proposed work employs acceleration structures over the elevation data to more efficiently find the intersection positions, most prominently cone maps [Dum06, PO06] and maximum-quadtrees [OKL06, TIS08]. However, all these methods are limited to individual height-field surfaces fitting into single texture resources.

Dick et al. [DKW09] describe a GPU-based ray casting approach able to handle arbitrarily large terrain data sets employing a multi-resolution quadtree representation of a tiled terrain elevation map. The height-field tiles corresponding to the nodes of the selected quadtree cut are rendered individually in front-to-back order applying a per-tile maximum-quadtrees. They account for occlusions between different height-field tiles during ren-

dering through the use of additional rendering passes prior to the actual ray casting in order to mask out already found height-field intersections. However, they do not account for data occlusions for the selection of the displayed quadtree cut. While they support extremely large elevation maps, their system is designed to distinctly handle single data sets.

For the handling of a fixed number of layered height fields, Policarpo et al. [PO06] proposed a single rendering-pass method derived from basic GPU-based ray casting approaches. This method, however, forgoes the use of any acceleration structure, strongly limiting its use cases. The method works by simply moving along the viewing-ray using a fixed sampling rate and intersecting all height fields at once using vector operations on the GPU. This works reasonably well for at most four height fields encoded in separate color channels of a single texture resource.

Regarding the application of existing rendering techniques to the visualization of large geological models, none of the existing ray casting-based rendering systems are capable of efficiently displaying complete stacks of highly detailed, partially translucent and, most importantly, mutually occluding and overlapping horizon height fields.

Design Goals

For a visualization system, supporting geological models containing multiple highly detailed horizon surfaces represented by large two-dimensional height-field primitives, we set the following design goals:

- Interactive visualization of multiple, large, mutually occluding and overlapping virtualized horizon height fields using a GPU-based ray casting approach.
- A single-pass rendering approach for entire horizon stacks, facilitating a correct visual representation of translucent horizon surfaces.
- Efficient ray-intersection computations in layered horizon height fields.
- Occlusion-aware level-of-detail selection.

In the following sub-sections, we present a ray casting-based rendering system for the visualization of geological subsurface models consisting of multiple highly detailed height fields. Based on our shared out-of-core data management system, we virtualize the access to the height fields, allowing us to treat the individual surfaces at different local levels of detail. The

visualization of an entire stack of height-field surfaces is accomplished in a single rendering pass using a two-level acceleration structure for efficient ray intersection computations. This structure combines a minimum-maximum quadtree for empty-space skipping and sorted lists of depth intervals to restrict ray intersection searches to relevant height fields and depth ranges. By utilizing the provided level-of-detail feedback mechanism, the updates of the multi-resolution hierarchy cuts of the individual height fields are performed inherently accounting for data visibility.

4.2.2 Acceleration Structure Considerations

Considering the absolute size of the height-field data sets our system is required to handle, the major challenge for a ray casting algorithm is an efficient ray traversal. For the implementation of our rendering approach, we considered two empty-space leaping methods: cone-step mapping introduced by Dummer [Dum06] and refined by Policarpo et al. [PO06], and maximum-quadtrees presented by Oh et al. [OKL06] and Tevs et al. [TIS08].

Cone-Step Mapping

Cone-step mapping is based on a cone ratio calculated for each height-field sample representing a circular cone. The opening angle of each cone describes the maximum angle for which it does not intersect the height field.

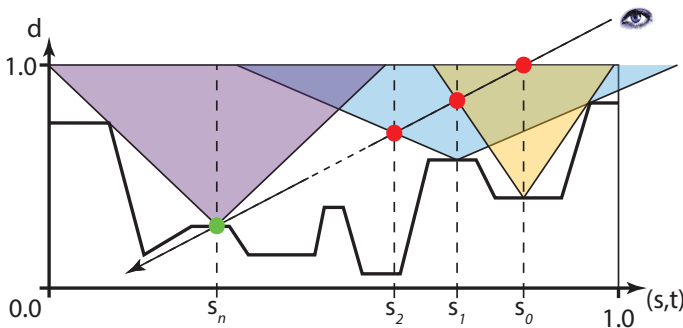


Figure 4.7: This figure illustrates the use of cone maps to accelerate ray-intersection searches on highly-detailed height-field data sets.

During ray traversal, the ray is intersected with the cone of the current sampling position. Then the sampling position is advanced to the calculated cone intersection, thereby skipping over larger regions of the height field. Figure 4.7 illustrates this ray-traversal process of a cone map defined in the two-dimensional (s, t) texture domain with d representing the depth or height values. The ray enters the height field at the sampling location s_0 . The next sampling position is determined as s_1 by the cone intersection at this position. This process repeats until the actual intersection s_n is eventually found. Cone-step mapping very quickly converges to the area of the ray intersections, but may require many steps to locate the actual intersection position as the computed cone intersections never actually intersect the height field. Policarpo et al. [PO06] improved upon the basic idea by relaxing the cone ratio to include height-field intersections under the constraint that a ray traversing a cone can only pierce the surface once. A successive application of a binary search between the last ray position and the cone intersection beneath the surface results in a more efficient ray-intersection estimation.

The application of cone-maps or relaxed cone-maps to very large horizon height-field surfaces, however, is impeded by two major drawbacks. Firstly, opposed to terrain visualization applications in which the actual height-field surface is only viewed from one side, horizons are potentially viewed from every angle. Therefore, for the application in seismic visualizations, two cone maps are required describing the cones on both sides of the surface. Assuming the storage of the cone ratios at the same precision as the height-field elevation values this approach introduces a 200% increase in data set size. The second drawback are extreme pre-processing times for the generation of the cone ratios for each height-field sample. The large memory overhead for the storage and the large pre-processing times to create the cone maps make its use for the visualization of multiple large horizons very inefficient. Further, the combined multi-resolution handling of the horizon height field associated with cone maps leads to potentially inconsistently stored cone ratios between different levels of detail, resulting in potential rendering errors.

Minimum-Maximum Quadtrees

Maximum-quadtrees, on the other hand, present a much reduced storage overhead for the acceleration structure while still allowing efficient ray-

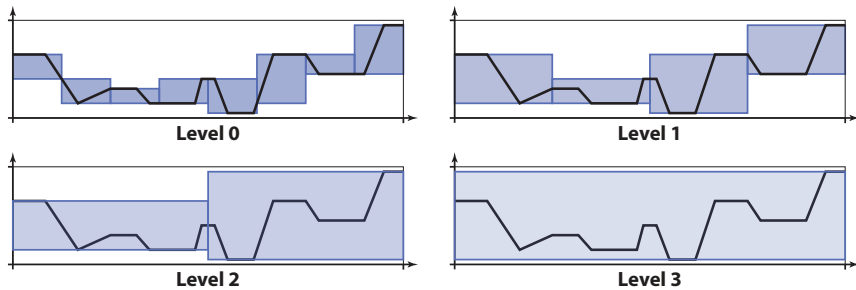


Figure 4.8: This figure illustrates the generation of a minimum-maximum quadtree over a tiled height field. The original data set is subdivided into fixed size tiles, representing the leaf nodes of the hierarchy. Each node is represented by the minimum and maximum values of its associated height-field tile. The quadtree hierarchy is created by recursively merging the values of four adjacent nodes on a finer level.

intersection searches. The GPU-based approach presented by Tevs et al. [TIS08] builds the quadtree hierarchy bottom-up starting with the combination of four adjacent height-field samples. They store the maximum value of all samples covered by the respective node. The quadtree hierarchy is created by recursively merging the values of four adjacent nodes on a finer level. Regarding horizon surfaces, this approach is easily extended to support two-sided height-field surfaces by storing an depth interval of the minimum and maximum values for each node resulting in a minimum-maximum quadtree, as illustrated in Figure 4.8.

The traversal of the viewing rays through the minimum-maximum quadtree is performed in top-down order. It starts by intersecting the ray with the root node of the hierarchy. The minimum-maximum intervals associated with each quadtree node are used to quickly discard nodes without possible intersections. When reaching a leaf node of the hierarchy, the stored node is checked for a ray intersection. This traversal is illustrated in Figure 4.9. If no intersection is found, the ray advances to the next node along the ray on the same quadtree hierarchy level. This approach leads to an inefficient traversal after the ray descended into the hierarchy but misses the surface. Tevs et al. [TIS08] describe a method to progressively ascend up the hierarchy in such cases, ensuring efficient subsequent ray traversals after close-miss situations.

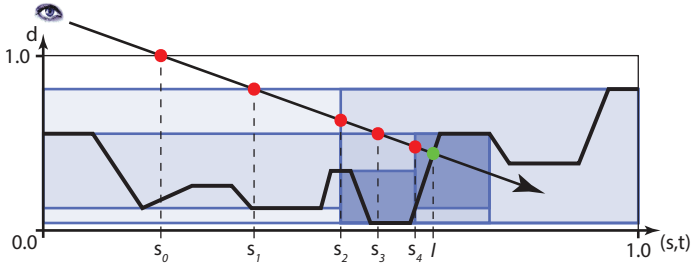


Figure 4.9: This figure illustrates the hierarchical traversal of a minimum-maximum quadtree over a tiled height field. The traversal recursively descends into the quadtree hierarchy upon detecting a ray intersection with the minimum-maximum interval stored in the tree nodes (red) until reaching a leaf node, which is then searched for the actual height-field intersection (green).

Both Oh et al. and Tevs et al. [OKL06, TIS08] store the quadtree hierarchy in the mipmap chain associated with the actual texture representing the height-field data on the GPU. Because they build the hierarchy based directly on the individual height-field samples, the direct application of their approach to large horizon height fields would result in very deep quadtree hierarchies. Furthermore, due to the multi-resolution representation of height fields in our rendering system, the maintenance of separate full quadtree hierarchies for the individual horizons poses a massive overhead. The multi-pass multi-resolution terrain rendering approach described by Dick et al. [DKW09] only applies a maximum-quadtree to the height-field tiles associated with the multi-resolution hierarchy cuts. Extending their approach to a single-pass method requires an additional acceleration structure for fast empty space leaping on the multi-resolution quadtree basis.

Considering layered height fields present in seismic models, currently no method exists preventing the simultaneous evaluation of multiple height fields during the ray traversal as described by Policarpo et al. [PO06].

4.2.3 Ray Casting of Stacked Virtualized Height Fields

The access to each horizon height field in our rendering approach is virtualized using our texture virtualization system. We utilize our level-of-detail

feedback mechanism to inherently resolve data occlusions in the horizon stacks (cf. Section 3.6). Therefore, occluded horizon parts can be represented at a much lower resolution than visible parts. This approach makes it possible to more efficiently balance the usage of the available memory resources among all horizon height fields. The selected multi-resolution cuts of the individual horizons are stored using the compact multi-resolution cut-hierarchy serialization as described in Section 3.4. The traversal of the serialization structure introduces logarithmic run-time overhead for the production of indirection information required for virtual texture coordinate translations, but maintains a very small memory footprint per height-field data set. Although the traversal routine is benefits from a good texture cache performance due to the small size and locality of the quadtree encoding, the cost of the tree traversal is not negligible. However, each ray typically samples the same height field tile multiple times and thus the traversal costs can be amortized over multiple height-field lookups.

Seismic models containing multiple stacked horizon surfaces possess a special property: the horizons are derived from a common seismic volume. Thus, the entire stack of horizon surfaces can be defined as equally sized height fields within the lateral (x-y) domain of the volume. The volume therefore acts as a frame of reference for the complete model and we are able to access the height values of different height fields at different levels of detail using the same global texture coordinates. Furthermore, the elevation values stored in the individual horizon height fields are stored in reference to their depth in the reference volume. Consequently, height samples from different horizons are directly comparable without any offset and scale operations.

Two-Level Acceleration Structure

In order to allow efficient traversal through such stacked data sets, based on different multi-resolution hierarchy cuts, we employ a two-level acceleration structure. A global minimum-maximum quadtree representing the primary structure for fast empty space skipping and a sorted list of height intervals associated with each quadtree node to restrict intersection searches to relevant horizons and depth ranges.

We restrict the size of the resulting top-level quadtree data structure by building the minimum-maximum hierarchy based on tiled horizon height fields. In fact, only a cut from this global quadtree is required during rende-

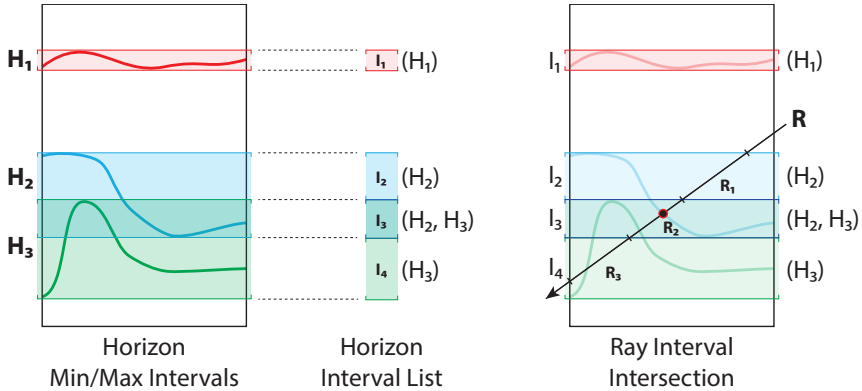


Figure 4.10: This figure shows the construction and traversal of the sorted depth-interval list for a single horizon height-field tile. The left image shows the original minimum-maximum intervals for three horizons. The sorted interval list with associated horizons is derived from the intersections of the source intervals. The right image shows the ray-intersection search in the successive intervals, resulting in the found intersection of the ray interval R_2 in the interval I_3 .

ring, which is a union of all the multi-resolution cuts of the individual height fields available on the GPU. The minimum-maximum ranges associated with each node are generated as the union of the individual ranges of the distinct height fields. However, we generate a sorted disjoint interval list for each quadtree node, which associates one or more horizons with each interval as shown in Figure 4.10. The resulting intervals are sorted by their minimum interval border in ascending order. The top-level quadtree is encoded in the mipmap hierarchy of a single texture for fast access and traversal on the GPU as described by Oh et al. and Tevs et al. [OKL06, TIS08]. Besides storing the minimum-maximum depth interval, each node additionally contains a link to its associated sorted horizon interval list.

Horizons typically contain areas of missing data or holes in fault regions. Therefore, individual horizon surfaces are described by two-dimensional partial height fields portraying scalar elevation values. For the distinction of holes in the horizons during a special value, tagging affected areas is employed in the height-field data set. This tag value is only regarded during

the rendering of the height-fields and is ignored during the pre-processing of the acceleration structures.

Ray Casting of Horizon Stack

The ray casting approach rendering an entire horizon stack is implemented in a single rendering pass. The processing of individual rays is triggered by the rasterization of the bounding geometry of the complete horizon stack, generating the ray-entry positions. The exit positions of the rays are determined analytically.

The traversal of minimum-maximum quadtree is performed in top-down order in a similar way as described by Oh et al. and Tevs et al. [OKL06, TIS08]. The minimum-maximum intervals associated with each quadtree node are used to quickly discard nodes without possible intersections. When reaching a leaf node in the current cut from the global quadtree, the actual horizons contained in the sorted horizon interval list are checked for intersections using a conventional combination of linear search and binary hit-point refinement. The evaluation order of the interval list is dependent on the actual ray direction. If the ray is ascending in the z-direction, the list is evaluated in front-to-back order starting with the interval with the smallest minimum value, while it is evaluated in back-to-front order for descending rays. If an intersection is found during the evaluation of an interval, it represents the closest height-field intersection and the evaluation process can be terminated early. In case of intervals associated with multiple horizons, we successively search all contained horizons for intersections in the appropriate order (cf. Figure 4.10). Once a horizon intersection is found in such an interval, we use the intersection point to restrict the subsequent searches in the remaining horizons. Due to the fact that the quadtree nodes represent horizon tiles containing 128×128 height values as one example, we perform a linear search along the ray within a tile to find a potential intersection interval followed by a binary search for finding the actual intersection point.

Upon finding a ray-surface intersection, we trigger the level-of-detail feedback mechanism by supplying it the found position in the texture space of the associated virtualized height field. The mechanism then determines the actually required height-field tile for the current viewer position as described in Section 3.6.2. The ray traversal is terminated by returning the found intersection position and color. In case of a surface intersection

with a translucent horizon, the ray traversal is continued right after an intersection is found and the shaded colors of all found intersections along the ray are composited in front-to-back order. The traversal is continued until the ray either leaves the horizon stack or the opacity is saturated, thereby terminating the ray traversal early.

Even though there is no restriction on the tile size used for the construction of the global minimum-maximum quadtree, using a tile size less than or equal to the tile size utilized by the multi-resolution quadtrees of the height fields allows for optimizations during the intersection search. Therefore, only a single query for the page atlas location of an associated height-field tile is required considering that all data lookups during the intersection search in a tile can be directly taken from this single atlas-texture sub-region. Thus, the traversal costs of the serialized height-field cuts for localizing a particular atlas page are amortized over a complete search interval, thereby significantly reducing the overhead of the virtualization approach.

Horizon Idiosyncrasies

Considering the two-sided nature of horizons exhibiting holes, we designed the ray traversal to allow for two-sided height-field surface intersections and discard false surface intersections belonging to tagged invalid data samples. Upon entering the bounding geometry of the horizon stack with a viewing ray, we classify if the ray is potentially piercing the surface from the top or the bottom. This classification is then used to detect the actual ray intersection event during the initial linear intersection search. However, through the special property of horizon height fields to potentially contain areas of missing data (e.g. in fault regions), this classification requires an update when the ray is passing through such an area and is therefore changing horizon sides.

The areas of missing data are encoded in our approach with the actual elevation value of 0, a value typically not occurring in horizon stacks as horizons are seldom tracked directly at the bottom of the seismic volume. Through the employed bilinear interpolation of the height-field samples, steep slopes around data holes are generated. The ray-intersection searches actually detect intersections with these slopes; however, we discard these false surface intersections during the ray traversal and therefore are able to generate sharp edges around fault regions or areas of missing data in the horizon surfaces. The detection of false intersections is based on the ability

of current GPUs to explicitly inspect the four values constituting a bilinear filtered sample. If only a single sample is detected to be the invalid data sample, the intersection is discarded and the ray traversal continued with the inverted surface-side classification.

Considering the multi-resolution representation of the individual height-field surfaces in the scene, certain artifacts may occur as the ray transitions between height-field tiles of different levels of detail. Particularly the occurrence of false cracks or holes in the surface may emerge through large differences in level-of-detail. However, through the tracking of the surface-side classification of the ray, such cases are prevented in our system.

4.2.4 Results

We implemented the described rendering system using C++. OpenGL4 and GLSL were used for the rendering related aspects and NVIDIA CUDA for the level-of-detail feedback evaluation. All tests were performed on a 2.8 GHz Intel Core i7 860 workstation with 8 GiB RAM equipped with a single NVIDIA GeForce GTX 480 graphics board running Windows 7 x64.

The rendering tests were performed with a data set containing a stack of three partly overlapping horizons provided to us by a member of the oil and gas industry. The dimensions of each height field defining a horizon surface are 6889×16984 points using 16 bit resolution per sample. The chosen page size for the virtualization of the height fields was 128^2 with a page overlap of two samples for the on-demand calculation of surface gradients. The employed page-atlas size was 128 MiB and the page-cache size was restricted to 1 GiB. The rendering tests were performed using a viewport resolution of 1680×1050 .

Our system is able to render scenes as shown in Figures 4.11 and 4.12 with interactive frame rates ranging from 20 Hz to 40 Hz depending on the viewer position, zoom level and choice of translucent display. We found that the rendering performance of our ray casting approach is mainly dependent on the screen projection size of the height fields and the chosen sampling rate. The memory transfers between the shared resources of the data-virtualization system have limited influence on the rendering performance. The level-of-detail feedback evaluation on the GPU introduces a small processing overhead below 2.5 ms per rendering frame in our testing environment. For the per-pixel feedback list generation, we utilized a list-

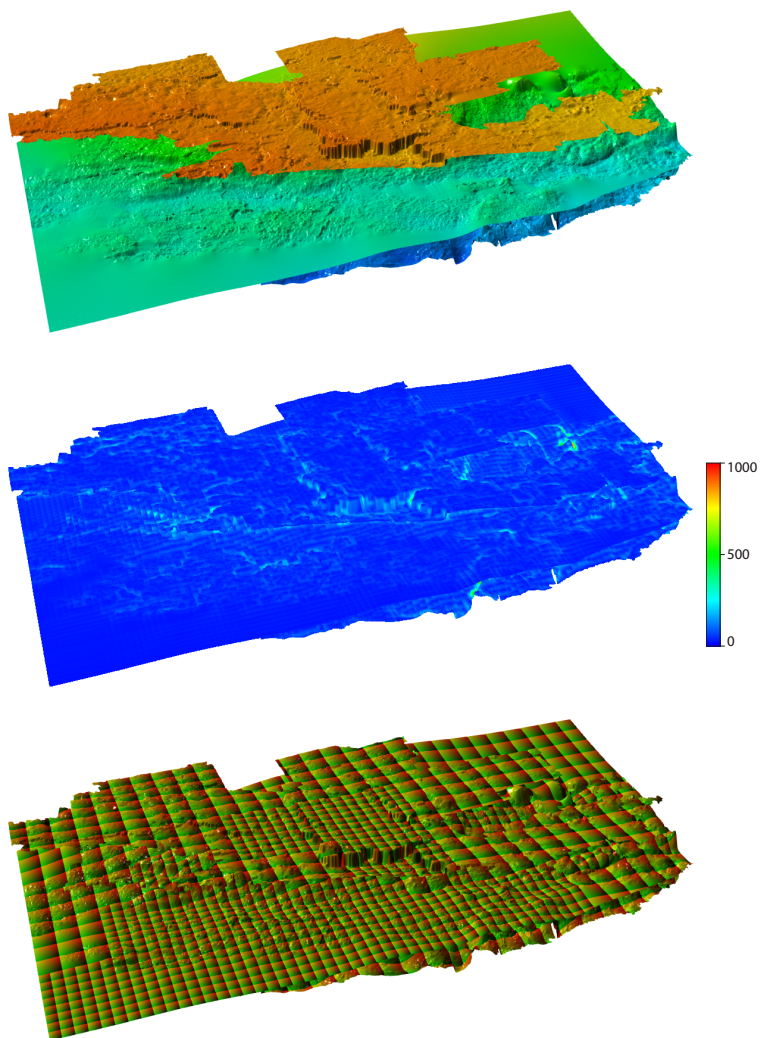


Figure 4.11: Example scene containing a horizon stack of three partially intersecting opaque surfaces. The images show (top to bottom) the final rendering of the horizons, the number of iteration steps of the ray traversal and the selected multi-resolution quadtree cut.

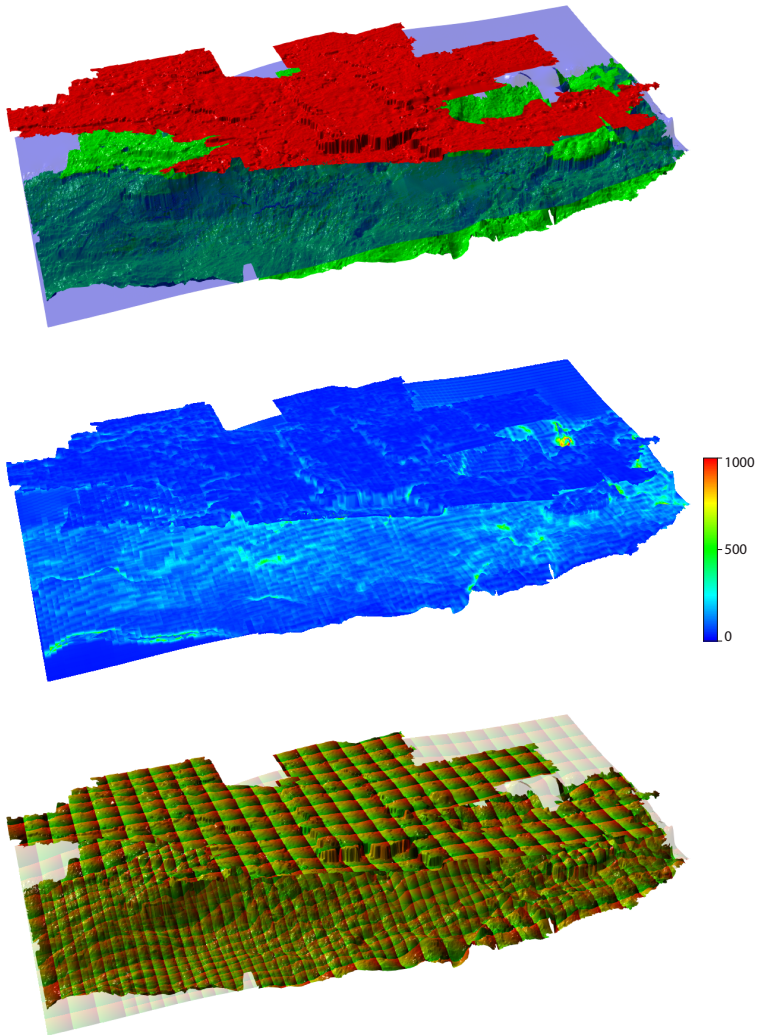


Figure 4.12: Example scene containing a horizon stack of three horizon surfaces with one horizon displayed translucently. The images show (top to bottom) the rendering of fully opaque horizons, the number of iteration steps of the ray traversal and the selected multi-resolution quadtree cut.

page size of four entries and a 4×4 sub-sampling of the viewport resulting in potentially 420×262 lists.

The comparison of the selected multi-resolution quadtree cuts is shown through the bottommost images. The height-field tiles are displayed using a color representation of their local coordinate system. It is evident that for the first case shown in Figure 4.11, in which all horizons are rendered completely opaque with the bottommost horizon largely occluded by the others, more resources are assigned to the visible height fields. The top-most horizon is especially represented by more smaller tiles. For the second case shown in Figure 4.12, in which the second horizon is displayed translucently, the bottommost horizon becomes mostly visible. Therefore, more shared resources are assigned to the now no longer occluded parts, withdrawing them from the other two. This becomes evident through the much larger height-field tiles used for the visualization of the topmost surface.

We evaluated various tile sizes for the construction of the global minimum-maximum quadtree ranging from 32^2 to 128^2 . The smaller tile sizes exhibit small improvements of 3-5% in rendering performance under shallow viewing angles compared to the largest size. However, under steep viewing angles, the introduced traversal overhead of the larger quadtree data structure results in a slight degradation of performance of 5-8%. The larger potential for empty space skipping using smaller tile sizes is canceled out by the complexity introduced by the depth interval list evaluation and the final intersection search in the height-field tiles. We found that using the same tile size for the global minimum-maximum quadtree as for the data-virtualization results in the best compromise between data structure sizes on the GPU and rendering performance. Figures 4.11 and 4.12 show that higher iteration counts occur in areas where the ray closely misses one horizon tile and intersects another one using a 128^2 tile resolution. Thus on the finest level, a ray needs to take an average of about 64 steps to find an intersection inside a tile assuming that such an intersection exists.

We also experimented with binary searches for finding the horizon intervals in a quadtree node. However, the limited number of actual depth intervals in our current data sets make this approach slower than simply searching linearly for the first relevant interval. For a larger number of stacked horizons, the binary search for the first relevant interval should improve performance.

Considering the caching of traversal information for the height-field virtualization mechanism during the ray-intersection search in a height-field tile, we found that an implicit caching mechanism implemented for the

virtual data look ups results in better run-time performance than an explicit approach transforming the ray intersection search to the local coordinate system of the height-field tile in the page atlas. This allows for a much clearer implementation of the ray casting algorithm decoupled from the actual multi-resolution data representation.

4.2.5 Conclusions

We presented a GPU-based ray casting system for the visualization of large stacked height-field data sets. Based on a shared out-of-core data management system, we virtualize the access to the height fields, allowing us to treat the individual surfaces at different local levels of detail. The multi-resolution data representations are updated using level-of-detail feedback information gathered directly during rendering. This provides a straightforward way to resolve occlusions between distinct surfaces without requiring additional occlusion culling techniques. The visualization of entire stacks of height-field surfaces is accomplished in a single rendering pass using a two-level acceleration structure for efficient ray intersection searches. This structure combines a minimum-maximum quadtree for empty-space skipping and sorted lists of depth intervals to restrict ray intersection searches to relevant height fields and depth ranges. A prototypical implementation shows that stacks of large height fields can be handled at interactive frame rates with mostly imperceptible loss of visual fidelity and moderate memory requirements.

4.3 Combined Rendering of Seismic Models

The visualization of geological models typically combines the display of several model components. Most seismic data sets feature a combination of interpreted horizon surfaces, their originating seismic volumes and other derived volumetric attributes. With the growing sizes of seismic surveys, the geological models contain volumetric primitives quickly growing beyond hundreds of gigabytes in size. The complexity of the derived horizon height fields is directly tied to the lateral size of the generating volumes, representing surfaces composed of several hundreds of million samples. With our out-of-core data management system, we are able to handle such large geological models with a fixed memory footprint. In the previous sections we presented rendering approaches utilizing our data-virtualization system to

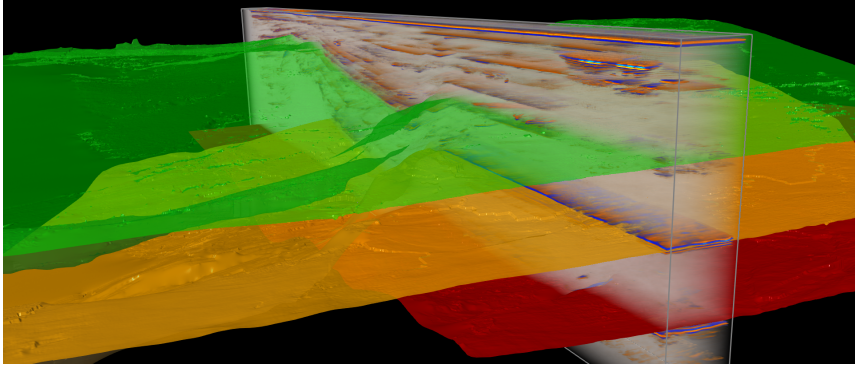


Figure 4.13: Exemplary visualization of a seismic model showing the combined rendering of partially translucent horizon surfaces and a portion of the seismic volume inside a volume lens.

efficiently render scenes composed of multiple large entities of a single type. In this section, we present a unified rendering approach for the combined handling of large seismic volumes and horizon stacks.

4.3.1 Problem Setting

With the demonstrated rendering approaches for large seismic volumes and large layered horizons, we already presented approaches capable of visualizing certain aspects of seismic models. A naïve combination of these approaches, however, will not generate optimal results with respect to rendering efficiency and the correctness of the visual results.

Regarding the visualization of a seismic model shown in Figure 4.13, the scene contains a volume lens visualizing a subsection of a seismic volume with an applied transfer function resulting in a translucent direct volume display. Two of the three contained horizon surfaces are also rendered translucently, enabling a view on the otherwise occluded parts of the scene. While most components of the scene are actually semi-transparent, large parts of the individual components are still invisible due to data occlusions. The chosen transfer function of the seismic volume tries to visually extract the most important reflection events as opaque regions while giving a semi-transparent context. As a result, in volume regions with a large amount

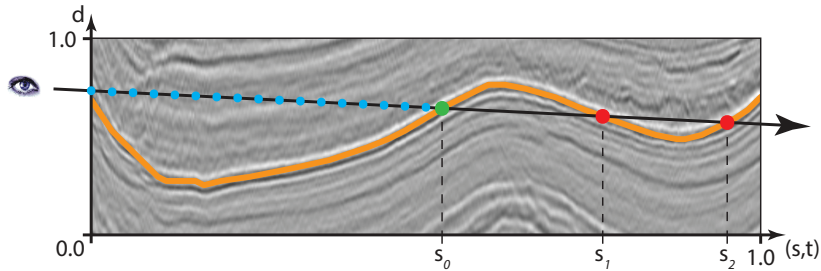


Figure 4.14: Naïve approach for the combination of translucent volume and horizon surface. The surface and volume primitives are rendered in separate rendering passes. The volume is accumulated to the first prior generated surface intersection (green), missing the integration of the volume segments determined by the further surface intersections (red).

of close reflection events, such as in the middle of the illustrated example, the volume visualization quickly accumulates to a non-transparent display, thereby covering parts of horizon surfaces. Conversely, most of the lower parts of the volumes are covered by the single opaque horizon.

With existing rendering solutions for the visualization of the individual components of a scene, current seismic visualization systems, and more general real-time rendering approaches, typically draw the surface and volume primitives in separate rendering passes and combine the results in the frame buffer. Thereby, the surface geometries are rendered first and the generated frame buffer content is employed by the subsequent volume rendering to limit the extent of the volume display. Figure 4.14 illustrates this approach for a single viewing ray on a volume ray casting approach. The horizon surface (orange) is rendered in a first rendering pass. Through the resulting frame buffer content for the particular viewing ray, the volume traversal is terminated when the first surface intersection (green) is reached. This approach generates correct rendering results for fully opaque surfaces. However, when using translucent surfaces, an inconsistent visualization is generated as no volume is visible through semi-transparent surfaces. The ray traversal is missing information about further surface intersections (red). Even an obvious extension, to separately integrate the volume in the front and in the back of the surface geometries through two frame buffer entries, will not generate correct visual results as multiple surface intersections

are still missed. With regard to our out-of-core data-virtualization system, depending on level-of-detail feedback generated during rendering, such approaches relying on separated rendering passes for individual scene parts would generate feedback entries even for eventually invisible scene parts.

Furthermore, besides the inconsistent visual display of translucent surface and volume primitives, a straightforward combination of the rendering of the two primitive types potentially leads to decreased rendering efficiency. While the previously described two-pass approaches make it possible to eliminate rendering overhead for volume parts occluded by horizon surfaces, the same is not the case in the reverse situation. If opaque volume parts occlude parts of horizon surfaces, the surface intersections are already processed. A naïve approach to solve this problem is a simultaneous ray traversal of the volume and height-field primitives. During this ray traversal, the volume and height-field data sets are sampled for each step along the ray to integrate the volume and concurrently check for surface intersections. Therefore, upon saturating the accumulated opacity along the ray, the calculation of an exact surface intersection can be prevented. However, such an approach is extremely inefficient as it forgoes empty space leaping optimizations for the surface intersection searches and therefore introduces large data sampling overhead.

Design Goals

For the design of our combined volume and layered height field rendering system, we are pursuing the following goals:

- ▶ Interactive visualization of a large horizon stack combined with its generating seismic volume.
- ▶ Single-pass rendering approach for entire geological models using a GPU-based ray casting method.
- ▶ Efficient combined ray casting of volume and height-field primitives preventing unnecessary rendering overhead related to occluded scene parts.
- ▶ Correct visual integration of translucent volume and height-field surface primitives.
- ▶ Occlusion-aware level-of-detail selection pertaining all scene components.

In the following sub-sections we present a rendering system capable of visualizing a geological model defined by a single seismic survey and derived volumetric and horizon surface primitives. The individual components of a large geological model are treated at varying levels of detail based on our out-of-core data-virtualization system. Data visibility is inherently taken into account through level-of-detail feedback collected directly during rendering, resulting in occluded scene parts being represented at a much lower resolution. The employed rendering method builds upon the efficient stacked horizon ray casting approach as described in Section 4.2. By integrating a volume ray traversal, we generate correct visual interaction between translucent height-field surfaces and translucent volume primitives. Furthermore, by exploiting the existing two-level acceleration structure used for the efficient ray traversal of the layered horizon surfaces, we terminate the surface-intersection search early in cases where accumulated volume samples occlude actual surface intersections, consequently increasing the rendering efficiency. As a result, the rendering system supports geological models consisting of large seismic volumes and multiple large height-field data sets, potentially exceeding the size of the graphics memory or even the main memory.

4.3.2 Ray Casting Virtualized Volumes

Before describing the combined rendering approach, we present an updated volume ray casting method for the rendering of virtualized large volumes. The volume ray casting method described in Section 4.1 lacks the ability to employ the advanced level-of-detail feedback mechanism available through our out-of-core data management system. In order to efficiently sample multiple arbitrary oriented volume primitives, the previously presented algorithm is based on full virtual volume data access during the generalized ray traversal. As we constrain the use of multiple volumes to a single volume grid, which acts as a frame of reference for all scene components, we can employ a more efficient volume ray casting method.

In order to reduce the per-sample data-lookup and virtual coordinate translation overhead present in the previous rendering approach, we employ a GPU-based ray casting method directly traversing the compact octree data structure. The rendering approach is based directly on the multi-resolution hierarchy cut serialization stored in GPU memory for the purpose

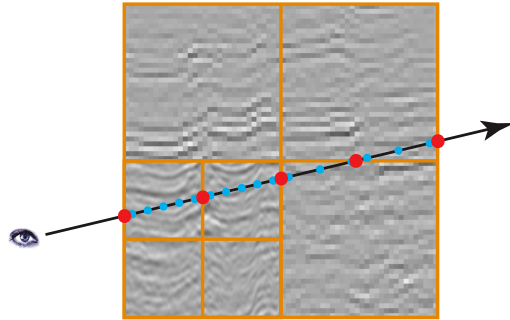


Figure 4.15: This figure illustrates the traversal of a viewing ray through a multi-resolution octree representation. The octree nodes intersected by the viewing ray are determined by a traversal of the underlying octree structure with the ray entry positions of the respective bricks (red). The traversal starts with the ray-entry position into the bounding volume. The actual volume integration (blue) is then performed locally in the volume bricks associated with the particular octree nodes.

of providing indirection information of volume data-pages stored in the employed page-atlas texture (cf. Section 3.4.2).

The ray casting approach is similar to the single-pass multi-resolution volume rendering methods presented by Gobbetti et al. [GMG08] and Crassin et al. [CNLE09]. The basic idea is to traverse the octree describing the current multi-resolution hierarchy cut, transform the ray into the physical coordinate system of a particular node and sample the stored volume brick in the atlas texture coordinate space, as illustrated in Figure 4.15. The major advantages of this approach are the reduced memory overhead for the storage of the hierarchy cut serializations, the drastically reduced overhead for virtual texture coordinate translations and the ability to implement adaptive sampling schemes based on the explicitly available level-of-detail information. The coordinate translation is performed only on a per-brick level. While the traversal of the octree serialization structure poses an increased overhead to produce the actual atlas-texture indirection information, this cost is amortized over the sampling of the volume-brick data directly in the physical atlas coordinate system.

A general implementation of the traversal of the octree hierarchy could be based on recursive DDA algorithms [AW87, Sun91]. Such approaches,

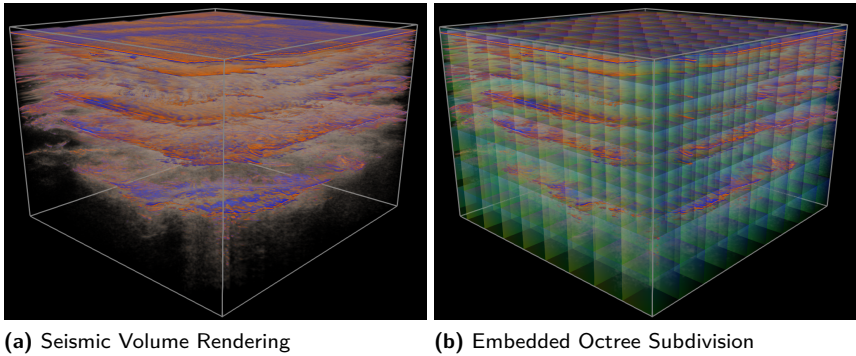


Figure 4.16: Visualization of a virtualized seismic volume based on a hierarchical volume ray casting approach. (a) shows the rendering results, while (b) illustrates the employed octree hierarchy representing the multi-resolution volume subdivision.

however, require stack data structures, which are currently not realizable in an efficient manner on today’s generations of GPUs. Gobbetti et al. [GMG08] employed pointers to neighboring nodes in the octree hierarchy for the progression from a particular node to the next along the ray. Our implementation uses an approach similar to *kd-restart*, which is typically employed for recursive tree-traversals in real-time ray tracing algorithms on the GPU [HSHH07]. Hence, the traversal algorithm utilized in our approach is very similar to the CPU-based octree traversal proposed by Glassner [Gla84]: The octree data structure is repeatedly traversed to determine all intersected octree nodes of the current octree subdivision during ray traversal. Such approaches forgo the reliance on stack structures or additional node links by restarting the hierarchy traversal when progressing into a new node.

Our single-pass volume ray casting approach is initiated by rendering the bounding geometry of the volume. This generates the ray-entry positions into the volume and therefore into the underlying multi-resolution octree hierarchy. The ray traversal starts by locating the particular octree node in the serialized tree-cut hierarchy. We can compute the actual extent and position of the node and perform an intersection with the viewing ray,

```

vec4
ray_cast_volume(
    in vtexture_3d vvolume, // virtual volume
    in ray          r,      // viewing ray
    in vec2         rseg,   // entry and exit ray parameters
    in float        d)      // sampling distance
{
    // ray parameter and position
    float t = rseg.x;
    vec3 p = r.origin + t * r.direction;
    // current octree node info
    node_info n = octree_traverse(vvolume, p);
    // output color variable
    vec4 dst = vec4(0.0);

    while ( t < rseg.y // inside volume bounds
           && dst.a < opacity_sat) { // early ray termination
        // append LOD request
        vtexture_append_request(vvolume, p);
        // calculate node bbox and exit position
        const bbox nbbox = node_bounding_box(n);
        const float t_exit = ray_box_intersect_exit(r, nbbox);
        const vec3 e = r.origin + t_exit * r.direction;
        // transform ray to atlas brick and integrate volume
        ray nr = transform_ray_to_brick(p, e, n);
        float nd = transform_dist_to_brick(d, n);
        integrate_brick_volume(vvolume, nr, nd, dst);
        // advance ray to next node
        t = t_exit;
        p = e;
        n = octree_traverse(vvolume, p);
    }
    return dst;
}

```

Listing 4.1: Pseudo GLSL-notation of the single-pass volume ray casting approach based on a virtualized volume texture.

resulting in the ray-exit position. With the indirection information obtained from the traversal, we then are able to transform the ray segment, defined by the entry and exit-position of the ray through the particular node, to the atlas-texture sub-region associated with the volume brick. The actual ray integration is performed directly in the volume brick without any further

coordinate translations. Upon finishing the processing of a node, the new ray position is used to determine the successive octree node and resulting atlas sub-region. In Listing 4.1, this compact virtual volume ray traversal scheme is shown using a pseudo code notation based on GLSL syntax. Through the iterative process, the entire multi-resolution hierarchy is traversed node by node. Figure 4.16 shows a rendering of a virtualized seismic volume and highlights the underlying octree data structure used during the ray traversal. In order to increase the rendering efficiency, we employ an adaptive sampling method which adjusts the sampling rate according to the actual level of detail of the processed current node.

The feedback requests to the level-of-detail feedback mechanism in our data-virtualization system are explicitly generated when entering a new octree node. Based on the employed texture-space metric, the system determines the actually required volume brick for the current viewer position and inserts the information into the associated feedback list as described in Section 3.6.2. Through the use of early ray termination upon saturating the accumulated opacity for a viewing ray, this approach inherently takes data occlusion into account. When selecting a transfer function largely rendering the volume opaque, feedback requests are only generated for the visible volume bricks on the surface of the bounding geometry (cf. Section 3.6.4). This makes it possible to balance the available volume data-page resources much more efficiently than the distance based metric used for the previous volume rendering approach.

4.3.3 Combined Ray Casting of Seismic Models

The definition of the combined rendering approach assumes the property of the seismic models that all components are defined in a shared coordinate system. The individual height-field surfaces contained in the horizon stack are derived from the same seismic volume, which therefore acts as a frame of reference for the entire scene. This property enables the definition of a single-pass ray casting algorithm traversing a seismic model as an extension to the previously presented rendering method for stacked horizon surfaces (cf. Section 4.2.3). The fundamental idea of our combined rendering approach is an iterative process determining a ray intersection with the horizon stack, integrating the volume up to this point and repeating this process until the ray leaves the bounding box of the model or until the accumulated opacity is saturated, facilitating early ray termination. However, as previously

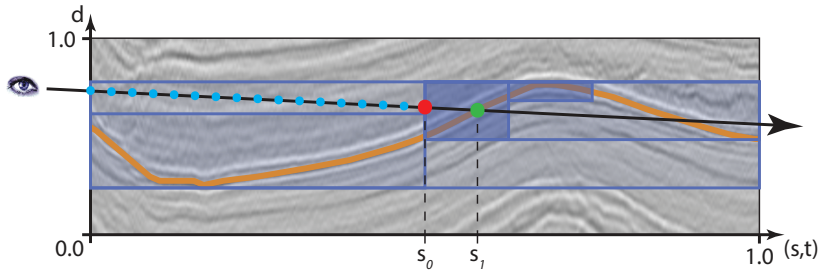


Figure 4.17: Single-pass rendering approach for the combination of a translucent volume and a horizon surface. Before determining the actual surface intersection at position s_1 (green) the volume is integrated up to the intersection with the minimum-maximum quadtree node at position s_0 (red). Only in the case of residual translucency will the method proceed with the surface-intersection search.

discussed, the determined surface intersection is often not actually visible because of opaque volume parts occluding the surfaces. In such cases unnecessary run-time overhead for the determination of redundant surface intersections should be avoided.

In order to reduce redundant surface intersections, we exploit the existing acceleration structure for the visualization of the stacked horizon height fields. Using the minimum-maximum quadtree and the linked depth interval lists, we can quickly determine an intersection position of the ray with the quadtree node of a height-field tile potentially containing a surface intersection. The volume is then accumulated up to this position. If, following this, the accumulated opacity along the ray is not saturated, we proceed to determine the actual surface intersection and continue the volume integration. This process is illustrated in Figure 4.17 for the first surface intersection along a single viewing ray. When actually entering a height-field tile, the algorithm consecutively processes all intersections in front-to-back order, as described in Section 4.2.3, and accumulates the volume contribution accordingly. At each step during this process, the ray traversal can be terminated early if the accumulated opacity is saturated. Upon leaving a height field tile, the method determines the next potential horizon height-field tile along the viewing ray and starts over with the coarse bounding geometry intersection. Listing 4.2 illustrates this basic staged single-pass rendering algorithm using

```

vec4
ray_cast_seismic_model(
    in vtexture_3d vvolume,          // virtual volume
    in vtexture_2d vhorizons[],      // virtual horizons
    in mm_quadtree mm_qtree,        // min/max quadtree structure
    in ray          r,              // viewing ray
    in vec2         rseg,           // entry, exit ray parameters
    in float        d)              // sampling distance
{
    // ray parameter
    float t = rseg.x;
    // output color variable
    vec4 dst = vec4(0.0);

    while ( t < rseg.y              // inside volume bounds
           && dst.a < opacity_sat) { // early ray termination
        // intersect min/max quadtree,
        // generating entry and exit parameters to hit node
        vec2 t_mm_node = intersect_minmax_qtree(mm_qtree, r, t);
        // integrate volume up to the node's entry position
        vec2 vol_seg = vec2(t, t_mm_node.x);
        integrate_volume(vvvolume, r, vol_seg, d, dst);
        // advance ray
        t = t_mm_node.x;
        // process actual ray-horizon intersections
        while ( t < t_mm_node.y      // inside min/max node
               && dst.a < opacity_sat) { // early ray termination
            // find closest horizon height field intersection
            vec4 col_hf;
            float t_hf = intersect_height_field_stack(
                vhorizons, r, t_mm_node, col_hf);
            // integrate volume up to the intersection position
            vol_seg = vec2(t, t_hf);
            integrate_volume(vtex_volume, r, vol_seg, d, dst);
            // integrate the horizon surface color
            composite_color(dst, col_hf);
            // advance ray
            t = t_hf;
        }
    }
    return dst;
}

```

Listing 4.2: Pseudo GLSL-notation of the single-pass ray casting algorithm for the combined visualization of a volume data set with embedded layered horizon height fields.

a GLSL-based pseudo code notation. The use of early ray termination upon saturating the accumulated opacity before actually intersecting a horizon surface prevents redundant intersection searches. In fact, only information available through the horizon stack acceleration structure are utilized in such cases, preventing costly virtual texture lookups to the horizon height fields.

The ray traversal is triggered by rendering the bounding geometry of the seismic volume defining the reference frame of the displayed seismic model. This generates the ray-entry positions for all viewing rays intersecting the model. In order to support a cubical volume lens, defining a sub-section of the seismic volume, we calculate the intersection of the individual viewing rays and the defining lens bounding box. This results in the actual ray-entry and exit parameters used to restrict the volume integration to specific ray segments. The volume rendering is using the ray casting approach presented in the previous section, which is based on a direct traversal of the serialized multi-resolution octree cut. The level-of-detail feedback mechanism is triggered for the volume when the ray is entering a new octree node in the current octree subdivision. For the height-field surfaces, level-of-detail requests are generated for actually processed surfaces intersections. Therefore, no requests are generated for surface regions covered by opaque volume areas and vice versa, resulting in the selection of coarser levels of detail in these data regions through the underlying data-virtualization system. This allows for more detailed data representations in actually visible data regions.

4.3.4 Results

We implemented the described rendering system using C++, OpenGL4 and GLSL as well as NVIDIA CUDA. The rendering related aspects of the system are based on OpenGL and the level-of-detail evaluation is based on a CUDA implementation. All tests were performed on a 2.8 GHz Intel Core i7 860 workstation with 8 GiB RAM equipped with a single NVIDIA GeForce GTX 680 graphics board running Windows 7 x64.

For the evaluation of our rendering system, we had access to two large seismic data sets. The first data set, from an oil field located in New Zealand, is defined by a volume with the dimensions $5989 \times 3933 \times 1501$ with a 32 bit floating point voxel precision, resulting in a raw volume size of 131.7 GiB. The data set contains a horizon stack composed of six layers with 16 bit

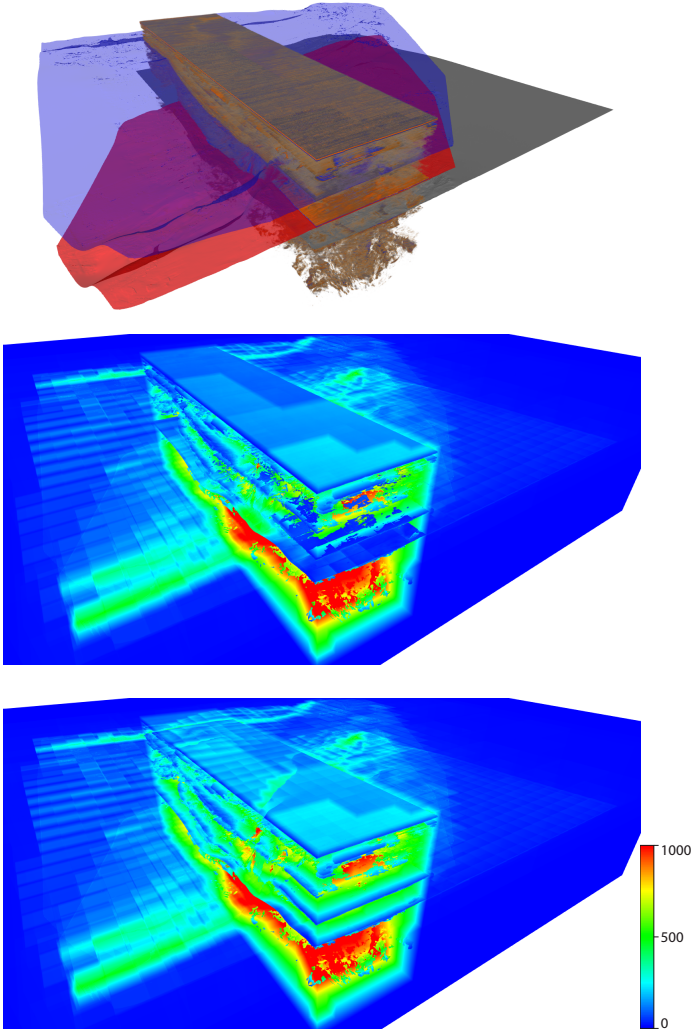


Figure 4.18: Example scenario 1: New Zealand data set with three horizon surfaces (the upper two displayed translucently) and a large volume lens. The lower two images show a comparison of the iteration steps for the combined rendering approach with and without the applied prevention of redundant surface intersection calculations.

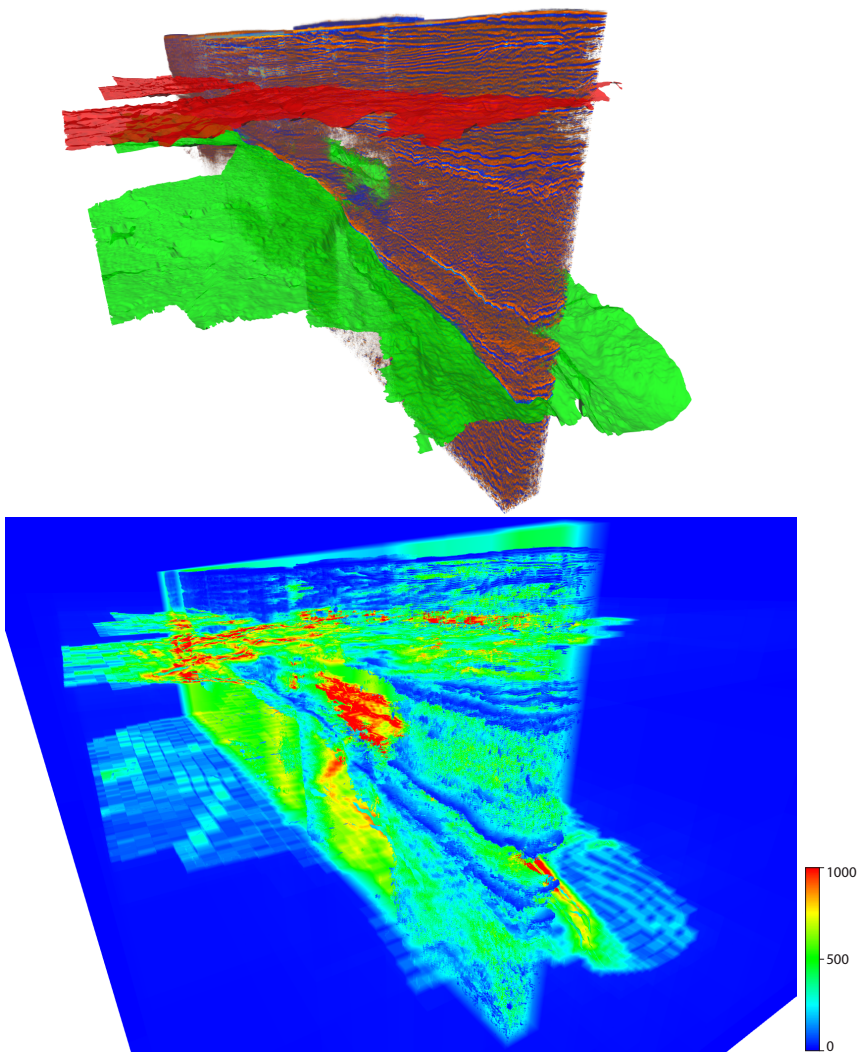


Figure 4.19: Example scenario 2: Norwegian data set with two semi-translucent horizon surfaces and a large volume lens. The bottom image shows the number of iteration steps of the ray traversal.

fixed point precision. The second data set, from a Norwegian oil field, is defined by a volume with the dimensions $6889 \times 16984 \times 1001$ at 8 bit fixed point voxel precision, resulting in a raw volume size of 109.1 GiB. The associated horizon stack contains three layers represented by height fields with 16 bit precision.

The employed rendering setup used a 1 GiB volume page-atlas texture and a 64 MiB height-field page-atlas texture to store the particular working sets, employing 64^3 volume bricks and 128^2 horizon tiles for the multi-resolution hierarchy representations of the respective data types. The rendering tests were performed using a viewport resolution of 1680×1050 and 4×4 sub-sampling for the feedback generation, resulting in a buffer of 420×262 potential feedback lists.

Our system is able to render the scenes as shown in Figures 4.18 and 4.19 with interactive frame rates ranging from 10Hz to 30Hz depending on the viewer position, zoom level and especially the chosen volume transfer function and selected horizon translucency. The rendering times directly depend on the viewport resolution and the screen projection size of the more costly volume components of the scene. The level-of-detail feedback evaluation process on the GPU introduces a processing overhead between 2 to 5 ms per rendering frame in the evaluation environment. The actual feedback evaluation overhead scales linearly with the amount of feedback request resources allocated (cf. Section 3.8.2). For the presented scenes, depending on the selected translucency of the volume primitives, we observed sizes of the dedicated feedback buffer ranging from 1 to 2.5 million entries.

A comparison of the ray iteration counts of the combined rendering approach with and without the utilization of the horizon stack acceleration structure is shown in Figure 4.18. It illustrates the effectiveness of the prevention of redundant surface intersections. The top part of the displayed volume, representing the sea floor in this data set, is opaque based on the chosen transfer function. While the ray traversal quickly stops in the area covered by the opaque volume parts when applying the optimization, an otherwise increased rendering overhead is observable related to the initial surface intersection. The real-world benefit of this ray traversal optimization again heavily depends on the visualization choices. We observed an up to 10% decrease in rendering times for the shown scenarios.

Finally, Figure 4.20 demonstrates the flexibility offered by our data-virtualization system. The visualization displays two volume attributes for the New Zealand data set. The originating seismic volume is shown through

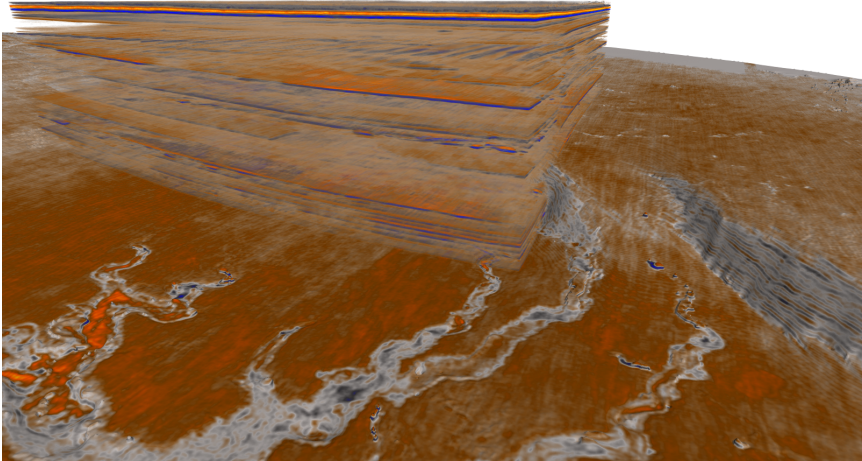


Figure 4.20: Example scenario 3: New Zealand data set with one horizon and two equally sized volume attributes. The first attribute is displayed directly through the volume lens, while the second attribute is mapped on an extracted horizon surface.

the large volume lens, while a second equally sized attribute is mapped directly to an extracted horizon surface. The data samples mapped to the surface are retrieved directly from the second virtualized volume through a virtual data lookup and are displayed using the same color mapping applied to the seismic volume. The required data blocks of the second volumetric attribute are requested only on the horizon surface. As a result, most of this volume is represented very coarsely with the higher level-of-detail data blocks only available in the areas coinciding with the horizon surface.

4.3.5 Conclusions

We presented a GPU-based ray casting approach for the combined visualization of large seismic volumes and derived horizon height-field surfaces, the most important elements of seismic models. The rendering system is based on an out-of-core data management system employing multi-resolution data representations for the individual components of a seismic model. The employed rendering method builds upon the efficient stacked horizon

ray casting approach described in Section 4.2. We show that a single-pass ray casting-based rendering approach facilitates correct visual interaction between translucent height-field surfaces and translucent volume primitives. Furthermore, by exploiting the existing two-level acceleration structure used for the efficient ray traversal of the layered horizon surfaces, we demonstrate the reduction of redundant intersections of occluded surfaces. The combination of a front-to-back ray casting approach and the direct level-of-detail feedback mechanism, provided by our out-of-core data-virtualization system, facilitates the selection of multi-resolution hierarchy cuts inherently accounting for data visibility. As a result, no additional occlusion culling techniques are required.

The prototypical implementation of the described out-of-core rendering system shows that large geological models, consisting of massive multi-attribute volumetric primitives and stacks of large height-field surfaces, can be visualized at interactive frame rates on commodity graphics hardware. The system operates with a fixed memory footprint for the storage of the current working sets of the handled data sets. We demonstrated how a flexible data-virtualization scheme allows for the realization of currently only difficult to achieve visualizations with respect to the handling and interaction of multiple potentially multi-attribute data sets.

4.4 Summary

The rendering algorithms presented in this chapter demonstrate the use of our out-of-core data-virtualization system in the specific application context of the visualization of large seismic models. Existing visualization systems struggle with the size and the amount of individual components in such data collections. The underlying multi-resolution data representations facilitate a level-of-detail handling of the individual data components, drastically reducing the size of the graphical working set of data required by the rendering algorithms. We demonstrate how efficient data virtualization allows for multi-resolution data sets to be treated in exactly the same way as regular data sets, thereby simplifying the definition of certain aspects of the rendering methods.

In particular, we present a volume ray casting method for rendering multiple arbitrarily overlapping multi-resolution volume data sets. We differentiate overlapping from non-overlapping volume regions by using a

BSP-tree based method. This approach provides a straightforward depth ordering of the resulting convex volume fragments and therefore avoids costly depth peeling procedures. On the basis of a full multi-resolution volume virtualization, each volume fragment resulting from the BSP subdivision is processed in a single ray casting rendering pass independent of the number of contained volume bricks.

The second rendering approach presents a ray casting method for the visualization of large stacked height fields representing horizon surfaces contained in seismic models. This approach employs a two-level acceleration structure for efficient ray intersection searches. This structure combines a minimum-maximum quadtree for empty-space skipping and sorted lists of depth intervals to restrict ray intersection searches to relevant height fields and depth ranges. We demonstrate the application of the level-of-detail feedback mechanism to the stacked horizon models facilitating the inherent consideration of data occlusions in scenes of high depth complexity without the application of costly external culling techniques.

The final presented rendering method combined the visualization of large seismic volumes and large stacked horizon height fields. The ray casting-based rendering method builds upon the efficient stacked horizon ray casting approach. We demonstrate that a single-pass rendering approach facilitates correct visual interaction between translucent height-field surfaces and translucent volume primitives, while simultaneously increasing rendering efficiency over traditional combined rendering approaches. This rendering system conveys the ability of our out-of-core data-virtualization system to handle multiple very large data components while using a fixed memory footprint. By exploiting feedback information about required levels of detail, we are able to deal with any possible form of data occlusion in a combined visualization.

Chapter 5

Conclusions

THROUGHOUT THIS THESIS, we explored algorithms and technologies for the unified handling and visualization of large-scale data sets. We particularly focused on real-time out-of-core data management and rendering methods for the visualization of extremely large geological and seismic data sets running on conventional computer systems. In the following sections, we will summarize our contributions and discuss possible future work in relation to the application of the presented techniques as well as directions and inspiration for future research.

5.1 Summary

Data Virtualization

The presented out-of-core data management and virtualization system is capable of simultaneously handling multiple large volume and height-field image resources. We demonstrate a unified data management system utilizing hierarchical multi-resolution data representations based on quadtree and octree data structures to generate level-of-detail working sets of the original data. In contrast to existing out-of-core data management systems which are specialized for single data sets of a particular data type, our system facilitates the simultaneous handling of multiple instances of different data types. System resources such as host and graphics memory as well as bandwidth resources are conceptually shared and balanced amongst all data types and data primitives.

Different levels of data abstraction are employed throughout the different stages of the system design. At the lowest level, the data sets are regarded as a set of data pages resulting from the initial subdivision of the original data sets. All data-scheduling decisions, such as the fetching of data from

external storage or the partitioning of a specific bandwidth budget for uploading data to the graphics memory are made without regard to the specific associated type of the data pages. In order to allow efficient access to the data on the GPU, this level of data abstraction is currently not achievable because of technical limitations of the current graphics hardware. On the GPU, the data sub-blocks associated with data pages are stored in atlas textures according to their specific type. As a result, the balancing of the memory resources on the GPU is dependent on the particular data type. This means that with the current system design, graphics resources are only shared among multiple entities of the same type (e. g. multiple volumes or height-field surfaces). However, this limitation does not affect the balancing of other important system resources such as the limited bandwidth between the external storage and the host system as well as between the main memory and the graphics memory.

The highest level of data abstraction in our system represents our approach for data-virtualization on the GPU. The data sets in graphics memory are represented by a set of data blocks described by cuts from the multi-resolution hierarchies of the original data sets. The indirection information, required to translate virtual sample locations to physical sample coordinates in the associated atlas textures, is provided by page tables or compact serializations of the multi-resolution hierarchy cuts describing the current working sets. The complexities of locating particular data blocks and the coordinate translation is hidden from potential application programmers. Consequently, the development of scientific visualizations is not affected by the size of the displayed data sets. Rendering algorithms, on the other hand, may utilize the multi-resolution hierarchy in order to more efficiently visualize large data sets. In these cases, our system allows access to the compact data representations on the GPU, such as for a single-pass volume ray casting algorithm directly utilizing the compact octree representation of a multi-resolution volume to find all volume sub-blocks intersected during ray traversal.

The working set generation in our system is based on a unified feedback system with inherent support for translucent geometric and volumetric data sets. This feedback system is jointly used for all data sets of the supported data types without special treatment of individual primitives. We show how per-pixel linked lists are used to store a varying number of level-of-detail requests for a subset of viewport pixels. The lists are filled directly during rendering with specific requests to data pages representing a certain level-of-

detail data block in the associated multi-resolution hierarchies. This allows us to inherently deal with data visibility without the application of additional costly and mostly complex to integrate occlusion culling techniques. As a result of the high depth complexity present in the visualization of seismic models, primarily caused by the display of translucent volume primitives, these feedback lists contain large amounts of information with a high degree of redundancy. This information is required by the hierarchy update mechanism running on the CPU. In order to reduce the amount of data required to be transferred from the GPU back to main memory as well as to reduce the run-time overhead for the evaluation of the feedback information, we employ a feedback compression phase executed directly on the GPU. We demonstrate how a histogram compression generates a compact feedback representation. This process counts the occurrences of individual requested data pages, which is used as a prioritization during the working set update process.

Rendering of Virtualized Seismic Data

Based on this out-of-core data-virtualization infrastructure, we present distinct rendering approaches for specific problem settings of the visualization of large seismic data sets. We demonstrate the application of the out-of-core data-virtualization system to this specific application area in order to handle geological models consisting of multiple large seismic volumes as well as large horizon height-field surfaces.

Multi-Volume Rendering We propose an efficient GPU-based volume ray casting system for the rendering of multiple arbitrarily overlapping multi-resolution volume data sets. Through the use of shared data resources in system and graphics memory, we are able to support a virtually unlimited number of simultaneously visualized volumetric data sets. We also demonstrate how the efficient volume virtualization allows for multi-resolution volumes to be treated exactly the same way as regular volumes. BSP volume decomposition of the bounding boxes of the cube-shaped volumes or volume lenses is used to identify the overlapping and non-overlapping volume regions. The resulting volume fragments are extracted from the BSP-tree in front-to-back order and rendered using a GPU-based single-pass ray-casting approach. We demonstrate how on the basis of a full multi-resolution volume virtualization, each volume fragment resulting from the BSP subdivision is processed independent of the number of contained volume bricks. This

presents the main advantage of our approach over similar work recently presented by Lindholm et al. [LLHY09], which bases the BSP construction on the individual sub-volume constituting the bricks of the multi-resolution hierarchy. As a result, our approach requires recomputations of the BSP tree only if the spatial relationship of the volumes changes.

Rendering of Layered Height Fields We further present a ray casting-based rendering system for the visualization of geological subsurface models consisting of multiple highly detailed height fields. Based on our shared out-of-core data management system, we virtualize the access to the height fields, allowing us to treat the individual surfaces at different local levels of detail (e.g. occluded horizon parts are represented at a much lower resolution). The visualization of an entire stack of height-field surfaces is accomplished in a single rendering pass using a two-level acceleration structure for efficient ray-intersection computations. This structure combines a minimum-maximum quadtree for empty-space skipping and sorted lists of depth intervals to restrict ray intersection searches to relevant height fields and depth ranges. The multi-resolution data representations are updated using the level-of-detail feedback information gathered directly during rendering. This provides a straightforward way to resolve occlusions between different surfaces without requiring additional occlusion culling techniques.

Combined Visualization of Volume and Layered Height Field Data We finally describe a rendering system for the combined visualization of entire geological models consisting of highly detailed stacked horizon surface geometries and massive volume data. Building on the previously presented stacked horizon ray casting approach, we demonstrate a single-pass rendering approach facilitating correct visual interaction between translucent height-field surfaces and volume primitives. Furthermore, we show how an integrated rendering approach for both graphical primitives can increase the rendering efficiency by preventing redundant intersection calculations for height-field surface areas occluded by accumulated volume contributions. Additionally, by employing the unified level-of-detail feedback mechanism, we inherently deal with occlusions between different data types in the combined visualization of volume and horizon surfaces.

With the thesis at hand, we demonstrate how a flexible data-virtualization system allows for the implementation of efficient rendering algorithms as well as the straightforward definition of currently only difficult to achieve

visualizations for such large data sets. The results obtained through prototypical implementations indicate that large seismic models can be handled at interactive frame rates with moderate memory requirements on currently available commodity graphics hardware.

5.2 Future Work

Technical Limitations and Implementation Choices

The design of the data-virtualization system proposed in this thesis offers unified out-of-core data management. The precious memory and bandwidth resources of the computer system are conceptionally shared among all data sets handled by the system. However, limitations of current generations of GPUs force the implementation of parts of the system specific to certain data types. Particularly, the implementation of typed page-atlas textures described in Section 3.4.1 is the result of the circumstance that texture resources are bound to a specific type and format. The resulting system design therefore allows only the sharing of GPU-memory resources between data sets of identical type. However, in the foreseeable future, GPUs will offer functionality specifically tailored to the support of virtualized texture resources.

One such feature is called *partially resident textures* [OvWS12] available in AMD's next-generation GPUs. This feature allows the definition of very large virtual texture resources in GPU memory. The most interesting aspect of this functionality is that independent of the type and format of these textures, the original resource is subdivided into and managed based on data-pages of a fixed memory size. The partially resident textures are accessed by special intrinsic functions facilitating transparent access to large resources without knowledge of the internal data structures, which is very similar to our system design. However, this functionality does not include the actual GPU-memory management. Upon trying to access a missing data page, the sample routines return error codes the applications is in turn required to handle. This means that the actual data management regarding the sharing of GPU-memory resources among different partially resident textures and the uploading of required data is still the responsibility of the rendering software. Furthermore, no inherent feedback mechanism is provided informing the application about missing parts of the partial

textures. Nevertheless, this functionality allows for a much more memory efficient implementation of our data-virtualization concept. The page atlas texture can be removed by assigning each data entity handled by the system a partially resident texture. Using this approach, the GPU memory can be truly shared among all data sets independent of their type and format. Our feedback mechanism is applicable to this functionality in order to communicate the missing pages back to the host to trigger the data uploads. The multi-resolution hierarchies currently employed by our system therefore need to represent data pages of the required fixed memory size. As a result, the data-virtualization system described in this thesis is adaptable to future hardware features providing improved flexibility for sharing GPU-memory resources among differently typed data sets.

Regarding the prototypical implementation of our system, we pointed out inefficiencies on the basis of current limitations of the available graphics and compute APIs. Especially limiting are context switches between the employed OpenGL and NVIDIA CUDA interfaces as described in Section 3.7.1. During the research conducted for this thesis, we observed the rise of alternative uses of GPUs for general purpose computations and thereby the introduction of specialized compute APIs. They often allow far more direct access to the GPU and its specialized features than the current graphics APIs such as OpenGL or Direct3D. For example, the facilities for direct access to independent DMA-units (direct memory access) [Nvi10] allow for less error-prone and effective implementations of out-of-core data management systems. Furthermore, currently the compute APIs offer more stable development environments and more advanced debugging facilities than the widely used cross platform graphics API OpenGL. From our current perspective, for a rendering system as described in this work, relying on complex data structures and ray casting-based rendering approaches, an implementation purely based on a modern compute API can provide vast advantages during the development as well as the actual system run-time.

Multi-Resolution Rendering Artifacts

A major factor currently preventing the widespread application of multi-resolution rendering approaches in the oil and gas domain are particular rendering artifacts inherent to the currently used approaches. Multi-resolution volume rendering approaches especially suffer from a very specific phenomenon. The octree hierarchies over the source data sets are generated by

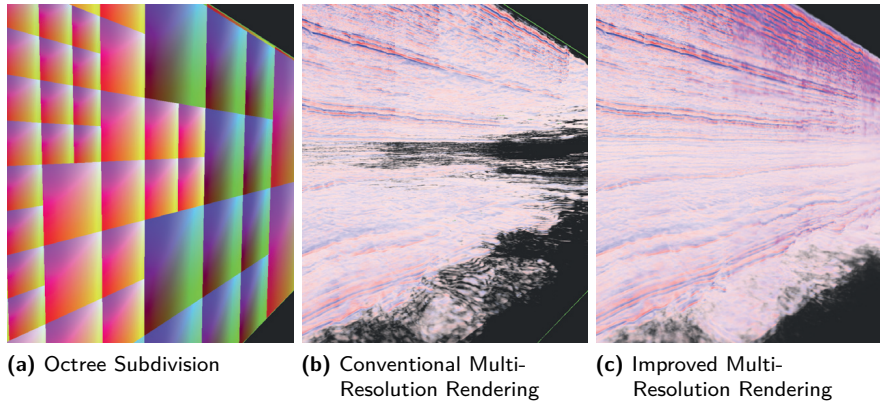


Figure 5.1: Comparison of conventional post-classification multi-resolution volume rendering and an approach generating the appearance of pre-classification rendering by approximating the data distribution on coarser levels of detail with a Gaussian basis function.

recursively filtering the high-resolution raw data values for each level of the hierarchy. During rendering, color and opacity values are determined by lookups to the transfer functions using the filtered data samples. Often these filtered values on coarser levels of detail generate completely different color and opacity values than expected. As a result, areas which contain visible structures based on the highest data resolution seem empty on coarser levels of detail. This is very apparent under the specific characteristics of seismic data sets with the data values of interest at the opposing sides of the value range (cf. Section 1.1). Figure 5.1 illustrates this problem for an exemplary octree subdivision of a seismic volume.

Younesy et al. [YMC06] proposed a method to improve the quality of multi-resolution visualizations. They approximated the data distribution of all data samples represented by each voxel at coarser levels of detail using a Gaussian distribution. This allowed them to pre-compute the influence of all represented data samples under the current transfer function and therefore more closely resemble the expected visual appearance in lower resolution regions of the multi-resolution volume. While Younesy et al. demonstrated their approach only for mipmapped volume data sets, we conducted first

experiments applying their approach to a full multi-resolution octree. The special characteristics of seismic volumes mostly adhere to the assumption of a Gaussian normal distribution for the data sets available to us. The achieved rendering results very closely resemble the expected rendering results as shown in Figure 5.1c.

We found that integration of the described approach requires minimal additional run-time overhead for the required extended transfer function lookup. Furthermore, due to the changed volume appearance to the more correct display of filled coarse volume regions, the early ray termination is more effective by allowing faster rendering times. Currently, the utilization of this technique is limited to 8 bit precision volumes and is affiliated with a memory overhead effectively doubling the memory required to store the additional parameters of the Gaussian distribution. The promising first results warrant further investigation for improved memory efficiency and the application to higher precision volume formats, which are becoming increasingly important in the oil and gas industry.

Time Varying Data

The out-of-core data management system presented in this thesis is designed to exclusively handle large static data sets. However, time varying data sets are becoming increasingly important in various application areas. In the oil and gas domain, series of seismic surveys are recorded during the production of an oil field in order to observe its development over time. Other sources of time varying seismic data sets are numerical simulations to predict hydrocarbon reservoir progression. The ability to efficiently visualize such unsteady volumetric data sets allows geo-scientists to observe the evolution of the underlying data over time in a more natural manner and opens up possibilities for new visualization methods. In order to efficiently handle time varying data sets, the requirements for the out-of-core data management system also increase by one additional dimension. For example, the working set selection algorithm needs to account for latencies when requesting the loading of data blocks of certain time steps ahead of time, or the level-of-detail feedback mechanism needs to incorporate some sort of predictive estimation of future required data blocks. We believe that our system design is flexible enough to be extended to support large time series of massive data sets.

Extended Rendering Approaches

A major limitation of the combined rendering approach, proposed in Section 4.3 of this work, is the constraint to a single reference volume grid, otherwise containing multiple volumetric attributes and derived horizon surfaces. With the multi-volume rendering approach described in Section 4.1, we demonstrated the use of binary space partitioning (BSP) to segment a scene of multiple arbitrarily overlapping multi-resolution volume grids into homogenous volume fragments. Similar approaches are conceivable for the application to multiple, arbitrarily overlapping seismic models. We believe that current generations of GPUs facilitate the application and direct traversal of hierarchical BSP subdivisions forgoing multi-pass rendering algorithms, and therefore shift more of the rendering algorithms to the specialized graphics hardware.

All rendering approaches proposed in this thesis are further limited to the sole visualization of large volumetric data sets and height-field surface geometries. While these account for the most important elements of seismic models, these data collections include additional large data sets of differing types. For example, reservoir body simulations generate large triangle meshes describing the outlines of hydrocarbon traps in the subsurface. Beyond that, highly detailed polygonal or parametric CAD models of oil production platforms or petroleum tankers are employed in virtual training or planning scenarios. We maintain that the correct visual integration of all the different data primitives contained in large seismic data collections is only achievable through ray casting or ray tracing-based rendering approaches. Currently no infrastructure exists for the efficient out-of-core management and rendering of regular geometry and volume data. Our ray casting-based approaches are an important step in this direction, since they facilitate the integration of recently presented GPU-based real-time ray tracing methods of polygonal data sets, such as the one presented by Horn et al. [HSHH07].

5.3 Outlook

The work presented in this thesis cannot suspend the growth of the data set sizes current visualizations systems are required to handle. However, we demonstrated techniques making it possible to efficiently manage and display data sets already multiple orders of magnitude larger in size than

the local memories of current computer systems. We believe that out-of-core data management and rendering algorithms will become increasingly important in various application areas. Effective virtualization of level-of-detail representations, abstracting the fact that at no point the data sets are entirely available at full resolution, in combination with output-sensitive rendering methods are major factors facilitating the process of developing efficient visualization systems for the most complex data sets.

Bibliography

- [AM00] U. Assarsson and T. Möller. Optimized View Frustum Culling Algorithms for Bounding Boxes. *Journal of Graphics Tools*, 5(1):9–22, January 2000.
- [AW87] J. Amanatides and A. Woo. A Fast Voxel Traversal Algorithm for Ray Tracing. In *Proceedings of Eurographics '87*, pages 3–10. Eurographics, 1987.
- [Bar08] S. Barrett. Sparse Virtual Textures. GDC 2008 presentations, <http://silverspaceship.com/src/svt/>, 2008.
- [BH11] N. Bell and J. Hoberock. Thrust: A Productivity-Oriented Library for CUDA. In W-M. W. Hwu, editor, *GPU Computing Gems Jade Edition*, pages 359–371. Morgan Kaufmann, October 2011.
- [BHMF08] J. Beyer, M. Hadwiger, T. Möller, and L. Fritz. Smooth Mixed-Resolution GPU Volume Rendering. In *IEEE/EG International Symposium on Volume and Point-Based Graphics*, pages 163–170. Eurographics, 2008.
- [BHP07] B. Brüderlin, M. Heyer, and S. Pfützner. Interviews3d: A Platform for Interactive Handling of Massive Data Sets. *IEEE Computer Graphics and Applications*, 27(6):48–59, November 2007.
- [BNS01] I. Boada, I. Navazo, and R. Scopigno. Multiresolution Volume Visualization with a Texture-Based Octree. *The Visual Computer*, 17(3):185–197, 2001.
- [BWPP04] J. Bittner, M. Wimmer, H. Piringer, and W. Purgathofer. Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful. *Computer Graphics Forum*, 23(3):615–624, 2004.

- [CCF94] B. Cabral, N. Cam, and J. Foran. Accelerated Volume Rendering and Tomographic Reconstruction using Texture Mapping Hardware. In *Proceedings of the 1994 symposium on Volume visualization*, pages 91–98. ACM, 1994.
- [CE97] M. Cox and D. Ellsworth. Application-Controlled Demand Paging for Out-of-Core Visualization. In *Proceedings of the 8th conference on Visualization '97*, pages 235–244. IEEE, 1997.
- [CE98] D. Cline and P. K. Egbert. Interactive Display of Very Large Textures. In *Proceedings of the conference on Visualization '98*, VIS '98, pages 343–350. IEEE, 1998.
- [CF11] R. Carmona and B. Froehlich. Error-controlled Real-Time Cut Updates for Multi-Resolution Volume Rendering. *Computers & Graphics*, 35(3):931–944, August 2011.
- [Cla76] J. H. Clark. Hierarchical Geometric Models for Visible Surface Algorithms. *Communications of the ACM*, 19(10):547–554, October 1976.
- [CN93] T. Cullip and U. Neumann. Accelerating Volume Reconstruction with 3D Texture Hardware. Technical Report TR93-027, University of North Carolina, Computer Science Department, 1993.
- [CN10] M. Cole and P. Norlund. High-resolution Displays Effective for Regional Interpretation. *E&P - Exploration and Production Magazine*, March 2010.
- [CNLE09] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann. Gigavoxels: Ray-Guided Streaming for Efficient and Detailed Voxel Rendering. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*, pages 15–22. ACM, 2009.
- [COCSD03] D. Cohen-Or, Y. L. Chrysanthou, C. T. Silva, and F. Durand. A Survey of Visibility for Walkthrough Applications. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):412–431, 2003.

- [Coo84] R. L. Cook. Shade Trees. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques (SIGGRAPH '84)*, pages 223–231. ACM, 1984.
- [CORLS96] D. Cohen-Or, E. Rich, U. Lerner, and V. Shenkar. A Real-Time Photo-Realistic Visual Flythrough. *IEEE Transactions on Visualization and Computer Graphics*, 2:255–265, September 1996.
- [Cra10] C. Crassin. OpenGL 4.0+ ABuffer V2.0: Linked Lists of Fragment Pages. <http://blog.icare3d.org/2010/07/opengl-40-abuffer-v20-linked-lists-of.html>, 2010.
- [CRF09] R. Carmona, G. Rodríguez, and B. Fröhlich. Reducing Artifacts Between Adjacent Bricks in Multi-Resolution Volume Rendering. In *Proceedings of the 5th International Symposium on Visual Computing, LNCS vol. 5876*, pages 644–655. Springer-Verlag, 2009.
- [CS93] D. Cohen and A. Shaked. Photo-Realistic Imaging of Digital Terrains. *Computer Graphics Forum*, 12(3):363–373, 1993.
- [CS99] W. Cai and G. Sakas. Data Intermixing and Multi-Volume Rendering. *Computer Graphics Forum*, 18(3):359–368, 1999.
- [DCH88] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume Rendering. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques (SIGGRAPH '88)*, pages 65–74. ACM, 1988.
- [DH92] J. Danskin and P. Hanrahan. Fast Algorithms for Volume Ray Tracing. In *Proceedings of the 1992 workshop on Volume visualization*, pages 91–98. ACM, 1992.
- [DKW09] C. Dick, J. Krüger, and R. Westermann. GPU Ray-Casting for Scalable Terrain Rendering. In *Proceedings of Eurographics 2009 - Areas Papers*, pages 43–50. Eurographics, 2009.
- [Don05] W. Donnelly. Per-Pixel Displacement Mapping with Distance Functions. In M. Pharr, editor, *GPU Gems 2*, pages 123–136. Addison-Wesley, 2005.

- [DSW09] C. Dick, J. Schneider, and R. Westermann. Efficient Geometry Compression for GPU-based Decoding in Realtime Terrain Rendering. *Computer Graphics Forum*, 28(1):67–83, 2009.
- [Dum06] J. Dummer. Cone Step Mapping: An Iterative Ray-Heightfield Intersection Algorithm. <http://www.lonesock.net/files/ConeStepMapping.pdf>, 2006.
- [DWS⁺97] M. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein. ROAMing Terrain: Real-Time Optimally Adapting Meshes. In *Proceedings of the 8th conference on Visualization '97*, pages 81–88. IEEE, 1997.
- [EHK⁺06] K. Engel, M. Hadwiger, J. Kniss, C. Rezk-Salama, and D. Weiskopf. *Real-Time Volume Graphics*. A.K. Peters, Ltd., Natick, MA, USA, 2006.
- [EKE01] K. Engel, M. Kraus, and T. Ertl. High-Quality Pre-Integrated Volume Rendering using Hardware-Accelerated Pixel Shading. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 9–16. ACM, 2001.
- [FAM⁺05] N. Fout, H. Akiba, K.-L. Ma, A. E. Lefohn, and J. Kniss. High-Quality Rendering of Compressed Volume Data Formats. In *Proceedings of The Joint EUROGRAPHICS-IEEE VGTC Symposium on Visualization 2005*, pages 77–84. IEEE, 2005.
- [FKN80] H. Fuchs, Z. M. Kedem, and B.F. Naylor. On Visible Surface Generation by a priori Tree Structures. In *Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pages 124–133. ACM, 1980.
- [Gal95] R.S. Gallagher. *Computer Visualization: Graphics Techniques for Scientific and Engineering Analysis*. CRC Press, Boca Raton, FL, USA, 1995.
- [GBAG04] S. Grimm, S. Bruckner, Kanitsarw A., and M. E. Gröller. Flexible Direct Multi-Volume Rendering in Interactive Scenes. In *Vision, Modeling, and Visualization (VMV)*, pages 386–379, 2004.

- [GKY08] E. Gobbetti, D. Kasik, and S. Yoon. Technical Strategies for Massive Model Visualization. In *Proceedings of the 2008 ACM symposium on Solid and physical modeling*, pages 405–415. ACM, 2008.
- [Gla84] A. S. Glassner. Space subdivision for Fast Ray Tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, 1984.
- [GM05] E. Gobbetti and F. Marton. Far Voxels: A Multiresolution Framework for Interactive Rendering of Huge Complex 3D Models on Commodity Graphics Platforms. In *ACM Transactions on Graphics (Proceedings of SIGGRAPH '05)*, pages 878–885. ACM, 2005.
- [GMG08] W. Gobbetti, F. Marton, and J. A. I. Gutián. A Single-pass GPU Ray Casting Framework for Interactive Out-of-Core Rendering of Massive Volumetric Datasets. *The Visual Computer*, 24(7-9):797–806, 2008.
- [GS01] S. Guthe and W. Straßer. Real-Time Decompression and Visualization of Animated Volume Data. In *Proceedings of the conference on Visualization '01*, pages 349–356. IEEE, 2001.
- [GS04] S. Guthe and W. Strasser. Advanced techniques for high quality multiresolution volume rendering. In *Computers & Graphics*, pages 51–58. Elsevier Science, 2004.
- [GWGS02] S. Guthe, M. Wand, J. Gonser, and W. Straßer. Interactive Rendering of Large Volume Data Sets. In *Proceedings of the conference on Visualization '02*, pages 53–60. IEEE, 2002.
- [GY98] M. E. Goss and K. Yuasa. Texture Tile Visibility Determination for Dynamic Texture Loading. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, HWWS '98, pages 55–60, New York, NY, USA, 1998. ACM.
- [HB04] M. Heyer and B. Brüderlin. Visibility-Guided Rendering for Visualizing Very Large Virtual Reality Scenes. In *Proceeding*

- 1st GI-Workshop on VR/AR TU-Chemnitz*, pages 163–172. Shaker Verlag, 2004.
- [HB05] M. Harris and I. Buck. GPU Flow Control Idioms. In M. Pharr and R. Fernando, editors, *GPU Gems 2*, chapter 34, pages 547–555. Addison-Wesley, March 2005.
- [HHS93] H.C. Hege, T. Höllerer, and D. Stalling. Volume Rendering - Mathematical Models and Algorithmic Aspects. Technical Report TR 93-7, Konrad-Zuse-Zentrum Berlin, 1993.
- [HKERS02] M. Hadwiger, J. M. Kniss, K. Engel, and C. Rezk-Salama. High-Quality Volume Graphics on Consumer PC Hardware. In *Course Notes 42 - SIGGRAPH '02*. ACM, 2002.
- [HP06] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, USA, 2006.
- [HPLVdW10] C. Hollemeersch, B. Pieters, P. Lambert, and R. Van de Walle. Accelerating Virtual Texturing Using CUDA. In W. Engel, editor, *GPU Pro: Advanced Rendering Techniques*, pages 623–641. A. K. Peters, Ltd., 2010.
- [HSHH07] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan. Interactive k-d Tree GPU Raytracing. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*, pages 167–174. ACM, 2007.
- [HSO07] M. Harris, S. Sengupta, and J. D. Owens. Parallel Prefix Sum (Scan) with CUDA. In H. Nguyen, editor, *GPU Gems 3*, pages 851–876. Addison-Wesley, August 2007.
- [Hüt98] T. Hüttner. High Resolution Textures. In *Proceedings of IEEE Visualization '98*, pages 13–17. IEEE, 1998.
- [JR97] J.-J. Jacq and C. Roux. A Direct Multi-Volume Rendering Method Aiming at Comparisons of 3-D Images and Models. In *IEEE Transactions on Information Technology and Biomedicine, Vol. 1*, pages 30–43. IEEE, 1997.

- [Jyl10] J. Jylänki. A Thousand Ways to Pack the Bin - A Practical Approach to Two-Dimensional Rectangle Bin Packing. <http://clb.demon.fi/files/RectangleBinPack.pdf>, February 2010.
- [Kau94] A. E. Kaufman. Voxels as a Computational Representation of Geometry. In *The Computational Representation of Geometry. SIGGRAPH '94 Course Notes*, page 45, 1994.
- [KBSS01] L. P. Kobbelt, M. Botsch, U. Schwanerke, and H.-P. Seidel. Feature Sensitive Surface Extraction from Volume Data. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques (SIGGRAPH '01)*, pages 57–66. ACM, 2001.
- [KD98] G. Kindlmann and J. W. Durkin. Semi-Automatic Generation of Transfer Functions for Direct Volume Rendering. In *In IEEE Symposium on Volume Visualization*, pages 79–86, 1998.
- [KE02] M. Kraus and T. Ertl. Adaptive Texture Maps. In *Proceedings of SIGGRAPH/EG Graphics Hardware Workshop '02*, pages 7–15. Eurographics, 2002.
- [KKH02] J. Kniss, G. Kindlmann, and C. Hansen. Multi-Dimensional Transfer Functions for Interactive Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):270–285, July 2002.
- [KSSE05] T. Klein, M. Strengert, S. Stegmaier, and T. Ertl. Exploiting Frame-to-Frame Coherence for Accelerating High-Quality Volume Raycasting on Graphics Hardware. In *Proceedings of IEEE Visualization '05*, pages 223–230. IEEE, 2005.
- [KW03] J. Krüger and R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings of IEEE Visualization 2003*, pages 38–43. IEEE, 2003.
- [LB98] L. A. Lima and R. Bastos. Seismic Data Volume Rendering. Technical report, University of North Carolina, Department of Computer Science, 1998.

- [LC87] W. E. Lorensen and H. E. Cline. Marching Cubes: A High Resolution 3d Surface Construction Algorithm. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques (SIGGRAPH '87)*, pages 163–169. ACM, 1987.
- [LC99] A. Leu and M. Chen. Modeling and Rendering Graphics Scenes Composed of Multiple Volumetric Datasets. *Computer Graphics Forum*, 18(2):159–171, 1999.
- [LCN98] B. Lichtenbelt, R. Crane, and s. Naqvi. *Introduction to Volume Rendering*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1998.
- [LDHJ00] E. LaMar, M. Duchaineau, B. Hamann, and K. Joy. Multiresolution Techniques for Interactive Texture-Based Rendering of Arbitrarily Oriented Cutting Planes. In *Proceedings EUROGRAPHICS/IEEE TVCG Symposium on Visualization 2000*, pages 105–114. IEEE, 2000.
- [LDN04] S. Lefebvre, J. Darbon, and F. Neyret. Unified Texture Management for Arbitrary Meshes. Technical Report RR5210-, INRIA, 2004.
- [Lev88] M. Levoy. Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988.
- [Lev90] M. Levoy. Efficient Ray Tracing of Volume Data. *Transactions on Graphics*, 9(3):245–261, 1990.
- [LH05] S. Lefebvre and F. Hornus, S. Neyret. Octree Textures on the GPU. In *GPU Gems 2*, pages 595–613. Addison-Wesley, 2005.
- [LH06] S. Lefebvre and H. Hoppe. Perfect Spatial Hashing. In *ACM SIGGRAPH 2006 Papers*, pages 579–588. ACM, 2006.
- [LHJ99] E. LaMar, B. Hamann, and K. I. Joy. Multiresolution Techniques for Interactive Texture-Based Volume Visualization. In *Proceedings of IEEE Visualization 1999*, pages 355–361. IEEE, 1999.

- [Lju06] P. Ljung. Adaptive Sampling in Single Pass, GPU-based Raycasting of Multiresolution Volumes. In *Proceedings Eurographics/IEEE International Workshop on Volume Graphics 2006*, pages 39–46. Eurographics, 2006.
- [LKR⁺96] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. A. Turner. Real-time Continuous Level of Detail Rendering of Height Fields. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques (SIGGRAPH '96)*, pages 109–118. ACM, 1996.
- [LLHY09] S. Lindholm, P. Ljung, M. Hadwiger, and A. Ynnerman. Fused Multi-Volume DVR using Binary Space Partitioning. In *Computer Graphics Forum, 28(3) (Proceedings Eurovis 2009)*, pages 847–854. Eurographics, 2009.
- [LLY06] P. Ljung, C. Lundström, and A. Ynnerman. Multiresolution interblock interpolation in direct volume rendering. In *Eurographics/IEEE-VGTC Symposium on Visualization 2006*, pages 259–266. Eurographics, 2006.
- [LP01] P. Lindstrom and V. Pascucci. Visualization of Large Terrains Made Easy. In *Proceedings of the conference on Visualization '01*, pages 363–371. IEEE, 2001.
- [LP02] P. Lindstrom and V. Pascucci. Terrain Simplification Simplified: A General Framework for View-Dependent Out-of-Core Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 8:239–254, July 2002.
- [Lue01] D. P. Luebke. A Developer’s Survey of Polygonal Simplification Algorithms. *IEEE Computer Graphics and Applications*, 21:24–35, 2001.
- [Max95] N. Max. Optical Models for Direct Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, June 1995.
- [Mor66] G.M. Morton. A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing. Technical report, Ottawa, Canada, IBM Ltd., 1966.

- [Mus88] F. K. Musgrave. Grid Tracing: Fast Ray Tracing for Height Fields. Technical Report RR-639, Yale University, Department of Computer Science, 1988.
- [Nad00] D. Nadeau. Volume Scene Graphs. In *Proceedings of the 2000 IEEE symposium on Volume Visualization*, pages 49 – 56. IEEE, 2000.
- [Nvi04] Nvidia. Improve Batching Using Texture Atlases. http://developer.download.nvidia.com/SDK/9.5/Samples/DEMOS/Direct3D9/src/BatchingViaTextureAtlases/AtlasCreationTool/Docs/Batching_Via_Texture_Atlases.pdf, July 2004.
- [Nvi10] Nvidia. NVIDIA Quadro Dual Copy Engines. http://www.nvidia.com/docs/IO/40049/Dual_copy_engines.pdf, October 2010.
- [Nvi12] Nvidia. NVIDIA CUDA C Programming Guide 4.2. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf, April 2012.
- [OBM00] M. M. Oliveira, G. Bishop, and D. McAllister. Relief Texture Mapping. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques (SIGGRAPH '00)*, pages 359–368. ACM, 2000.
- [OKL06] K. Oh, H. Ki, and C.-H. Lee. Pyramidal Displacement Mapping: a GPU based Artifacts-free Ray Tracing Through an Image Pyramid. In *Proceedings of the ACM symposium on Virtual reality software and technology, VRST '06*, pages 75–82. ACM, 2006.
- [OvWS12] J. Obert, J. M. P. van Waveren, and G. Sellers. Virtual Texturing in Software and Hardware. In *ACM SIGGRAPH 2012 Courses*, pages 5:1–5:29. ACM, 2012.
- [Owe07] J. D. Owens. Towards Multi-GPU Support for Visualization. *Journal of Physics: Conference Series*, 78(1):012055 (5pp), 2007.

- [PG07] R. Pajarola and E. Gobbetti. Survey on Semi-regular Multiresolution Models for Interactive Terrain Rendering. *The Visual Computer*, 23:583–605, July 2007.
- [PGSF04] J. Plate, A. Grundhöfer, B. Schmidt, and B. Fröhlich. Occlusion Culling for Sub-Surface Models in Geo-Scientific Applications. In *Joint Eurographics - IEEE TCVG Symposium on Visualization*, pages 267–272. IEEE, 2004.
- [PHF07] J. Plate, T. Holtkaemper, and B. Froehlich. A Flexible Multi-Volume Shader Framework for Arbitrarily Intersecting Multi-Resolution Datasets. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1584–1591, 2007.
- [PO06] F. Policarpo and M. M. Oliveira. Relief Mapping of Non-Height-Field Surface Details. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 55–62. ACM, 2006.
- [POC05] F. Policarpo, M. M. Oliveira, and J. L. D. Comba. Real-time Relief Mapping on Arbitrary Polygonal Surfaces. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, I3D '05, pages 155–162. ACM, 2005.
- [PTCF02] J. Plate, M. Tirtasana, R. Carmona, and B. Fröhlich. Octreemizer: A Hierarchical Approach for Interactive Roaming Through Very Large Volumes. In *Proceedings of the Symposium on Data Visualisation 2002*, pages 53–60. IEEE, 2002.
- [QQZ⁺03] H. Qu, F. Qiu, N. Zhang, A. Kaufman, and M. Wan. Ray Tracing Height Fields. In *Proceedings of Computer Graphics International*, pages 202–207, 2003.
- [RBE08] F. Roessler, R. P. Botchen, and T. Ertl. Dynamic Shader Generation for Flexible Multi-Volume Visualization. In *Proceedings of IEEE Pacific Visualization Symposium 2008 (PacificVis '08)*, pages 17–24. IEEE, 2008.
- [RGW⁺03] S. Roettger, S. Guthe, D. Weiskopf, T. Ertl, and W. Strasser. Smart Hardware-Accelerated Volume Rendering. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '03*, pages 231–238. IEEE, 2003.

- [RS05] T. Randen and L. Sønneland. Atlas of 3d seismic attributes. In A. Iske and T. Randen, editors, *Mathematical Methods and Modelling in Hydrocarbon Exploration and Production*, volume 7 of *Mathematics in Industry*, pages 23–46. Springer-Verlag, Berlin, Heidelberg, Germany, 2005.
- [RSEB⁺00] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive Volume on Standard PC Graphics Hardware using Multi-Textures and Multi-Stage Rasterization. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 109–118. ACM, 2000.
- [RTF⁺06] F. Roessler, E. Tejada, T. Fangmeier, T. Ertl, and M. Knauff. Gpu-based multi-volume rendering for the visualization of functional brain images. In *Proceedings of SimVis 2006*, pages 305–318, 2006.
- [SA12] M. Segal and K. Akeley. The opengl graphics system: A specification (version 4.3). www.opengl.org/documentation/specs, 2012.
- [SCESL02] C. T. Silva, Y.-J. Chiang, J. El-Sana, and P. Lindstrom. Out-of-Core Algorithms for Scientific Visualization and Computer Graphics. In *Visualization '02 Course Notes*, 2002.
- [Sch05] H. Scharlach. Advanced GPU Raycasting. In *Proceedings of the 9th Central European Seminar on Computer Graphics*, pages 69–76, 2005.
- [SG95] R. E. Sheriff and L. P. Geldart. *Exploration Seismology*. Cambridge University Press, Cambridge, UK, 1995.
- [SMG03] P. M. Silva, M. Marcos, and M. Gattass. 3D Seismic Volume Rendering. Technical report, Pontifical Catholic University of Rio de Janeiro, 2003.
- [SSKE05] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A Simple and Flexible Volume Rendering Framework for Graphics-Hardware-based Raycasting. In *Proceedings of Volume Graphics 2005*, pages 187–195. Eurographics, 2005.

- [Sun91] K. Sung. A DDA Octree Traversal Algorithm for Ray Tracing. In *Proceedings of Eurographics '91*, pages 73–85. Eurographics, 1991.
- [TIS08] A. Tevs, I. Ihrke, and H.-P. Seidel. Maximum Mipmaps for Fast, Accurate, and Scalable Dynamic Height Field Rendering. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 183–190. ACM, 2008.
- [TMJ98] C. C. Tanner, C. J. Migdal, and M. T. Jones. The Clipmap: A Virtual Mipmap. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques (SIGGRAPH '98)*, pages 151–158. ACM, 1998.
- [Vit01] J. S. Vitter. External Memory Algorithms and Data Structures: Dealing with Massive Data. *ACM Computing Surveys*, 33(2):209–271, 2001.
- [WE98] R. Westermann and T. Ertl. Efficiently Using Graphics Hardware in Volume Rendering Applications. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques (SIGGRAPH '98)*, pages 169–177. ACM, 1998.
- [Wil83] L. Williams. Pyramidal Parametrics. In *Proceedings of the 10th annual conference on Computer graphics and interactive techniques (SIGGRAPH '83)*, pages 1–11. ACM, 1983.
- [WKME03] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-Based Ray Casting for Tetrahedral Meshes. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, pages 44–. IEEE, 2003.
- [WM95] W. A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.
- [WMG98] C. M. Wittenbrink, T. Malzbender, and M. E. Goss. Opacity-Weighted Color Interpolation for Volume Sampling. In *Proceedings of the 1998 IEEE symposium on Volume visualization*, pages 135–142. ACM, 1998.

- [WWH⁺00] M. Weiler, R. Westermann, C. Hansen, K. Zimmerman, and T. Ertl. Level-of-Detail Volume Rendering via 3D Textures. In *Proceedings of the 2000 IEEE Symposium on Volume Visualization*, pages 7–13. IEEE, 2000.
- [YHGT10] J. C. Yang, J. Hensley, H. Grün, and N. Thibieroz. Real-Time Concurrent Linked List Construction on the GPU. *Computer Graphics Forum*, 29(4):1297–1304, 2010.
- [YHN07] B. Yost, Y. Hacıahmetoglu, and C. North. Beyond Visual Acuity: The Perceptual Scalability of Information Visualizations for Large Displays. In *CHI’07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 101–110. ACM, 2007.
- [YMC06] H. Younesy, T. Möller, and H. Carr. Improving the Quality of Multi-Resolution Volume Rendering. In *Proceedings of Eurographics/IEEE-VGTC Symposium on Visualization 2006*, pages 251–258. Eurographics, 2006.
- [YS93] R. Yagel and Z. Shi. Accelerating Volume Animation by Space-Leaping. In *Proceedings of the 4th conference on Visualization ’93*, pages 62–69. IEEE, 1993.

Curriculum Vitae

Personal

Name : Henning *Christopher* Lux
Birth Date : July 3, 1979
Birth Place : Arnstadt/Thuringia, Germany

Education

1986–1991 : Rudolf-Teichmüller-Oberschule, Ichtershausen
1991–1998 : Neideck-Gymnasium · Staatliches Gymnasium I, Arnstadt
1999–2004 : Diploma study in Computer Science at Technische Universität Ilmenau, Faculty of Computer Science and Automation

Professional

2004–2005 : Working Student, Virtual Reality Competence Center, DaimlerChrysler Research Center, Ulm
2005–2012 : Research and Teaching Associate with the Virtual Reality Systems Group, Faculty of Media, Bauhaus-Universität Weimar
2012–present: Senior Graphics Software Engineer, NVIDIA Advanced Rendering Center GmbH, Berlin

Book Chapters

- C. Lux. The OpenGL Timer Query. In *OpenGL Insights*, Editors: Patrick Cozzi and Christophe Riccio, A K Peters Ltd./CRC Press, 2012.

Conference and Journal Publications

- J.P. Springer, C. Lux, D. Reiners, and B. Fröhlich. Advanced Multi-Frame Rate Rendering Techniques. In *Proceedings IEEE Virtual Reality 2008 Conference*, pages 177–184. IEEE, 2008.
- D. Kurz, C. Lux, J.P. Springer, and B. Fröhlich. Improved Interaction Performance for Ray Tracing. In *Proceedings of Eurographics 2008*, pages 283–286. Eurographics, 2008.
- A. Kulik, A. Kunert, C. Lux and B. Fröhlich. The Pie Slider: Combining Advantages of the Real and the Virtual Space. In *Proceedings of the 10th international Symposium on Smart Graphics*, LNCS vol. 5531, pages 93–104. Springer-Verlag, 2009.
- C. Lux, and B. Fröhlich. GPU-based Ray Casting of Multiple Multi-Resolution Volume Datasets. In *Proceedings of the 5th International Symposium on Visual Computing*, LNCS vol. 5876, pages 104–116, Springer-Verlag, 2009.
- A. Kunert, A. Kulik, C. Lux, and B. Fröhlich. Facilitating System Control in Ray-based Interaction Tasks. In *VRST '09: Proceedings of the 16th ACM Symposium on Virtual Reality Software and Technology*, pages 183–186, ACM, 2009.
- M. Reichl, R. Dünger, A. Schiewe, T. Klemmer, M. Hartleb, C. Lux and B. Fröhlich. GPU-based Ray Tracing of Dynamic Scenes. In *Journal of Virtual Reality and Broadcasting 7(2010)*, no. 1, GI VR/AR Workshop 2008 Special Issue, 2010.
- C. Lux, and B. Fröhlich. GPU-based Ray Casting of Stacked Out-of-Core Height Fields. In *Proceedings of the 7th International Symposium on Visual Computing*, LNCS vol. 6938, pages 269–280, Springer-Verlag, 2011.

Ehrenwörtliche Erklärung

ICH erkläre hiermit ehrenwörtlich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Weitere Personen waren an der inhaltlich-materiellen Erstellung der vorliegenden Arbeit nicht beteiligt. Insbesondere habe ich hierfür nicht die entgeltliche Hilfe von Vermittlung- bzw. Beratungsdiensten (Promotionsberater oder anderer Personen) in Anspruch genommen. Niemand hat von mir unmittelbar oder mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt.

Ich versichere, dass ich nach bestem Wissen die reine Wahrheit gesagt und nichts verschwiegen habe.

Berlin, den 12. April 2013

— Christopher Lux —